UCRL-WEB-201535

DPCS/LCRM Reference Manual

Table of Contents

Preface 4
Introduction 5
Background 6
DPCS Architecture 8
- Resource Allocation and Control System (RAC) 9
  - RACCOM (RAC Communications Daemon) 10
  - ACC (Report Accounts) 11
  - BAC (Report Bank Names and Privileges) 12
  - BT (Defunct) 13
  - RA (Defunct) 13
  - NEWACCT (Set Current Account) 13
  - DEFACCT (Set Default Account) 14
  - RACMGR (RAC Manager Daemon) 15
- Production Workload Scheduler (PWS) 16
  - The PWS Daemons (PWSD, PLSD, BCD) 16
  - The PWS User Utilities 18

DPCS Operating Features 19
- Status Values for Batch Jobs 19
  - Interpreting Status Values 19
  - Alphabetical List of Status Values 20
- Class Values for Batch Jobs 26
- Run Properties of Batch Jobs 27
- Resource Partition Limits 30
- Environment Variables for Batch Jobs 32
- Comment and Shell Handling 35
- Job Scheduling 37
  - Order of Checking Precluding Conditions 37
  - Algorithm for Job Scheduling 39
- Output Truncation 43
- Reporting Memory and Time Used 44
- Log Files for Done Jobs 45
- DFS and DCE Interactions with Batch 46
- Managing Nonshareable Resources 46
- Expediting and Exempting Jobs 47
  - Expediting Jobs 48
  - Exempting Jobs 50
  - Forcing Job Priorities 51
  - Granting Special-Job Permissions 52

Fair Share Scheduling Algorithms 54
- Definitions 54
  - Shares 54
  - Active Users 55
- Shares and their Normalization 56

Usage and Its Decay 58
Priority Calculation 60
Role of Priority in Job Scheduling 63
Graceful Priority-Service Transition 64
Warning Alternatives 64
Library Calls 65
PCSGETRESOURCE (LRMGETRESOURCE) 65
PCSSIG_REGISTER (LRMSIG_REGISTER) 67
PCSWARN (LRMWARN) 68
Error Conditions (*pcsstatus) 69
Examples 70
Poll-for-Warning Examples 70
Signal-Catching Examples 75
Administrative Examples 80
Checkpointing 81
Checkpointing Overview 81
Condor Automatic Checkpoint 81
Program-Generated Checkpoint 82
A DPCS Resubmitting Script 86
Disclaimer 90
Keyword Index 91
Alphabetical List of Keywords 93
Date and Revisions 95

Preface

Scope: The DPCS/LCRM Reference Manual explains in detail the role, architecture, components, operating features and behavior, and typical applications of LC's Distributed Production Control System (often loosely called the "batch system"). The software that DPCS uses to manage batch jobs (both user utilities and hidden daemons) and the effect of DPCS management on those jobs are described at length. One chapter explains the concepts, terms, and formulas that comprise "fair-share scheduling" as implemented on LC machines. Other chapters tell how to use the Condor libraries to support voluntary checkpointing, and how to gracefully handle unexpected batch job terminations.

Readers interested in step-by-step instructions for making a batch script and running it should instead consult the EZJOBCONTROL (URL: http://www.llnl.gov/LCdocs/ezjob) guide. General usage-reporting and limit-reporting tools are covered and illustrated in the Bank and Allocation Manual (URL: http://www.llnl.gov/LCdocs/banks). Specific details about gang scheduling of parallel jobs appear in the Gang Scheduler User Guide (URL: http://www.llnl.gov/LCdocs/gang).

Starting in 2003, DPCS began changing its official name to the "Livermore Computing Resource Management" (LCRM) system, although its user messages and most user utilities remain unchanged.

Availability: When the programs described here are limited by machine, those limits are included in their explanation. Otherwise, they run under any LC UNIX system.

Consultant: For help contact the LC customer service and support hotline at 925-422-4531 (open e-mail: lc-hotline@llnl.gov, SCF e-mail: lc-hotline@pop.llnl.gov).

Printing: The print file for this document can be found at

```
OCF: http://www.llnl.gov/LCdocs/dpcs/dpcs.pdf
SCF: https://lc.llnl.gov/LCdocs/dpcs/dpcs_scf.pdf
```

Introduction

The Distributed Production Control System (DPCS) allocates resources for the UNIX-based production computer systems at Lawrence Livermore National Laboratory (LLNL). Through a complex hierarchy of computer-share bank accounts, job limits on banks and users, time-usage monitoring tools, and run-control mechanisms, DPCS lets organizations control who uses their computing resources and how rapidly those resources are used. It also manages an underlying batch system that actually runs production jobs guided by DPCS policies.

Thus DPCS both delivers computing resources to LLNL's scientists and provides for accurate accounting of resource use to government oversight agencies. Its uniform interface lets all production machines be managed as one, which reduces operating costs. And organizations control their own budgeted allocations (e.g., the way compute shares divide among users), which reduces the active involvement of the computer center.

This reference manual for DPCS describes the many software daemons and user utilities that comprise the system and shows how they are related. Relevant status messages, environment variables, and other operating features are explained as well. Pitfalls or unexpected side effects of DPCS features or algorithms are noted and explained throughout the text, as well as how to handle unexpected changes in job status. Future editions will also include nonstandard batch techniques, such as using RUN with PROXY and managing parallel batch jobs.

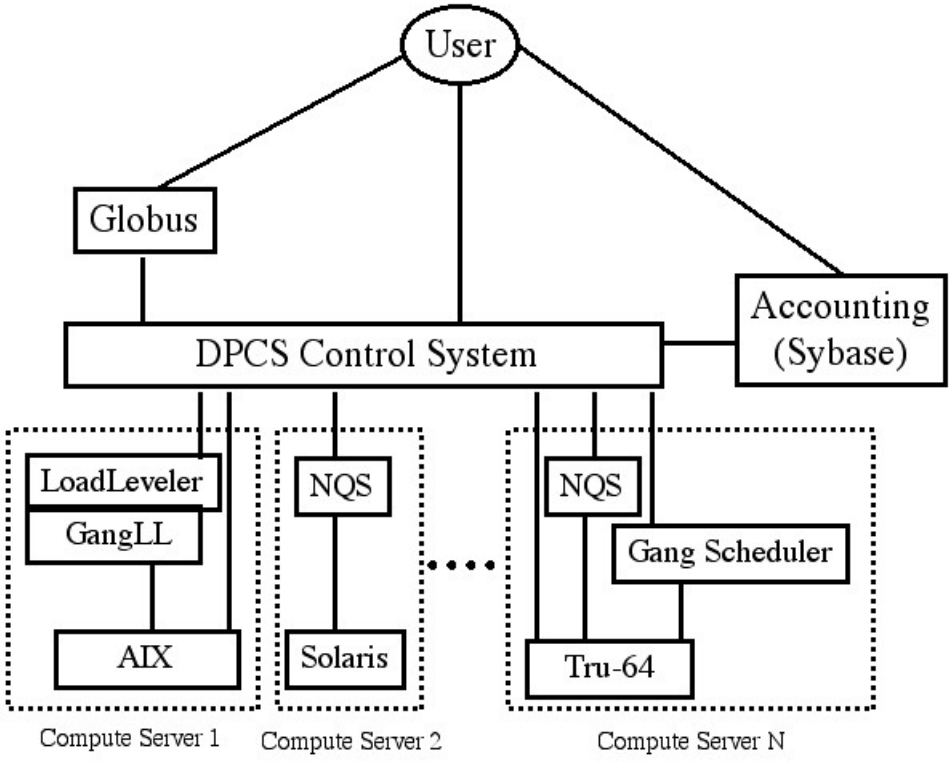
This is not a basic user guide to batch processing. For such a step-by-step primer of usage information please see EZJOBCONTROL (URL: http://www.llnl.gov/LCdocs/ezjob). For reference information on how to monitor computer time and its use (or job limits and their commitment so far), see the Bank and Allocation Manual (URL: http://www.llnl.gov/LCdocs/banks) (the Bank Manual also contains explicit instructions for allocating time for those few LC users authorized to manage resource banks).

Starting in 2003, DPCS began changing its official name to the "Livermore Computing Resource Management" (LCRM) system. This means that internal names, structures, files, and libraries have changed from "pcs" to "lrm" (example: the API library LIBPCS.A became LIBLRM.A). But user messages still mention DPCS and user tools retain their original names (exception: former utility PCSMGR can now only be executed as LRMMGR, and its interactive prompt has become lrmmgr> instead of the former pcsmgr>). Also starting in 2003, LC began deploying a locally designed, low-level resource manager to work "below" DPCS/LCRM (from a user's viewpoint) to more efficiently handle nodes and tasks for large parallel jobs. See the SLURM Reference Manual (URL: http://www.llnl.gov/LCdocs/slurm) for details on what this low-level system contributes to job control, especially on Linux (CHAOS) machines.

Background

In 1990, Livermore Computing (LC) committed to convert its production platforms to UNIX-based systems. This was a radical change because we had developed and come to rely on elaborate accounting and resource management facilities in our earlier, proprietary systems.

The DPCS project, begun in 1991 and in operation since October, 1992, adapts production demand to present an efficiently manageable workload to the kernel memory and CPU schedulers by monitoring memory load, swap device load, and idle time. The DPCS is not a CPU scheduler or a low-level batch system (it relies on other underlying batch systems, such as NQS, LoadLeveler, or LC's own SLURM (URL: http://www.llnl.gov/LCdocs/slurm)). Nor does it do process-level, process-termination accounting.



DPCS features include:

- Basic data collection and reporting mechanisms for project-level, near-real time accounting.
- Resource allocation to customers according to customers' organizational budget.
- Automated, highly flexible system with feedback for proactive delivery of resources.
- Flexible prioritization of production, including "run on demand."
- Dynamic reconfiguration and retuning.
- Graceful degradation in service to prevent overuse of the machine where not authorized.

- Proactive delivery of service to organizations that are behind in their consumption of resources to the extent possible via the underlying batch system.

In the mid 1990s, the LC staff extended DPCS to support massively parallel processing (MPP) architecture. With this upgrade, DPCS is able to schedule production jobs that span a large number of tightly coupled homogeneous processing elements.

DPCS has also been extended to support clustered machines. To schedule a job on a clustered machine, a user only needs to specify the cluster name (or a computing feature that only resides on the cluster) rather than any particular node in the cluster.

LC has also extended DPCS to allow cross-host submission of production jobs to any of several heterogeneous platforms from any platform. Support has been added so that allocations and production scheduling is managed from a single platform for all hosts. Further, the entire DPCS system can be managed from any host rather than each host being managed locally.

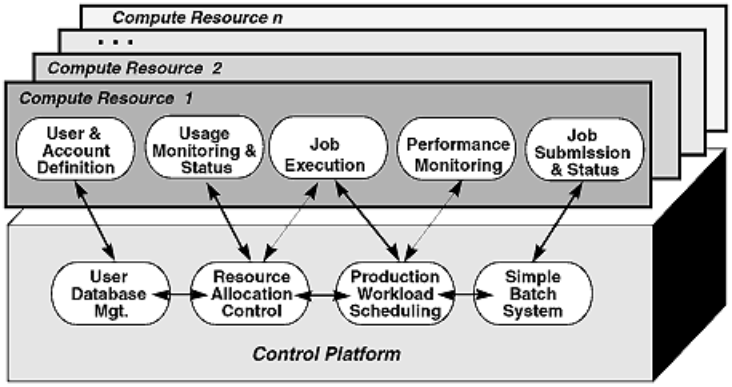
DPCS manages jobs on Solaris, AIX, Digital UNIX (Compaq's TRU64), and Linux/CHAOS. The DPCS "central managers" use IBM high-availability machines, computers with redundant processors and disks with automatic fault recovery, for maximum reliability.

Starting in 2003, DPCS began gradually changing its official name to the "Livermore Computing Resource Management" (LCRM) system. Most system messages continue to refer to DPCS, however, and most system user utilities retain their original names. The exception is the former PCSMGR utility, which you must now execute as LRMMGR (and which now offers an lrmmgr> prompt instead of a pcsmgr> prompt). The API library LIBPCS.A has become LIBLRM.A as well.

The maximum length of a DPCS job name is 15 characters, but user names may be as long as 31 characters.

DPCS Architecture

DPCS consists of two major parts, shown in the lower center of the figure below. The Resource Allocation and Control (RAC) subsystem allocates resources to organizations and controls access to those resources. The Production Workload Scheduler (PWS) schedules production computing jobs (batch jobs) on machines to efficiently deliver resources as desired.



Resource Allocation and Control System (RAC)

The RAC system manages job behavior through:

- Recharge accounts (charge account number; not the same as "login" account or user).
- Banks (allocation pools or group).
- User allocations within the banks.

As resources are consumed on a machine, the RAC system associates them with the user and the appropriate bank and recharge account. A user and the user's bank are debited, and a report is made to charge the account in near-real time. Under fair-share scheduling, users and banks are allocated shares that control the *rate* at which they consume computing resources rather than the total *amount* of resources they consume. Nevertheless, time accounting as managed by the RAC system continues to reveal important time-used trends retrospectively.

Historically, at LC an account was essentially a credit that represented an amount of usable resources (which may be unlimited). Some users, called account coordinators, were permitted to manage the account by granting and denying other users access to it. Accounts were independent of each other; that is, accounts had no subaccounts. Accounts were used primarily to determine budgetary charges for organizations and to provide users with a "one stop" limit on resource accessibility. Currently, however, LC accounts are used ONLY for voluntary time-used record keeping, NOT for allocation delivery. See the Bank and Allocation Manual (URL: http://www.llnl.gov/LCdocs/banks) for the current comparative role of accounts, banks, and allocations.

Banks manage the rate of delivery of allocated resources, and prioritize and manage production on each machine. A bank represents a resource pool (not of time, but of nondecremented shares) available to sub-banks and to users who have access permission. Banks exist in a hierarchical structure: one "root" bank "owns" all the resources on a machine, which are apportioned to its sub-banks. There is no limit to the depth of the hierarchy. Some users, called bank coordinators, may create and destroy sub-banks and grant and deny other users access to a bank. The authority of coordinators extends from the highest level bank at which they are named coordinator throughout that bank's subtree. Users access part or all of a bank's resources through one or more user allocations (sometimes constrained by per-bank or per-user job limits (page 30)).

The following subsections explain the software components of the RAC system, which include both daemons and user utilities.

RACCOM (RAC Communications Daemon)

The RAC system communications daemon (RACCOM) reads messages from RAC clients (any of six user utilities related to accounting), forwards the messages to RACMGR (page 15), reads the reply from RACMGR, and forwards it back to the client.

In addition, RACCOM is the parent process for the daemons RACMGR, RACRPT, and PWSD (page 16) (having been started up as a result of an exec by pcsstart). As such, it monitors the health of these processes. It logs shutdowns and critical failures and shuts down the PCS in case of critical error.

There are six "native" RACCOM client utilities (some now defunct), each of which has its own section including user instructions and typical usage examples below. NOTE: for many kinds of time-used reports it is more appropriate to run the PCSUSAGE utility (described and illustrated in the Bank and Allocation Manual (URL: http://www.llnl.gov/LCdocs/banks)). The RACCOM clients are:

ACC	reports currently available account numbers.
BAC	reports your bank name and whether you can use short-production (now obsolete) or expedited runs.
BT	(defunct) reported your allocated, used, and available bank time. See its section for replacements.
RA	(defunct) reported your allocations by shift (day, night, weekend). See its section for replacements.
NEWACCT	changes the current account that your usage draws against.
DEFACCT	changes the default account that your usage draws against.

ACC (Report Accounts)

To display DPCS acount permissions and account numbers, use the ACC command. (NOTE: ACC is mostly disabled because account use became voluntary on all machines. For retrospective time-used reporting, you should run PCSUSAGE (URL: http://www.llnl.gov/LCdocs/banks) instead.) Typical usage examples for ACC include:

```
acc
     Shows all the accounts that exist on the
     machine where it is run.  Remember that account
     use is voluntary and NOT associated with
     allocations of computer time.  [All previous
     ACC options are now disabled.]
```

BAC (Report Bank Names and Privileges)

The BAC command is used to display access information from the RAC database. BAC reports were modified (fall, 2003) to display full bank names and user names up to 31 characters. Typical examples of BAC usage include:

```
bac  report the access status of the calling user to his/her  current
     bank

bac -u joan,steve,mary
     report the access status of users steve, joan and  mary  to  all
     their banks

bac -b sab,fll
     report the access status of the banks sab and fll.

bac -t fll -l 3
     report the access status of all child nodes of  fll  down  three
     generation levels.

bac -l 3 -T root
     report the access status of the top three levels of the rdb  but
     report only banks.  Do not report user allocations.

bac -t root -0
     report the access status of all banks and user allocations  from
     which some time has been used.

bac -T root -0
     report the access status of all banks from which some  time  has
     been used.  Do not report user allocations.

bac -r sab
     report the access status of bank sab and all of its  parents  up
     through the root bank.

bac -u joan -b sab
     approximately the same as:  bac -b sab; bac -u joan
```

BT (Defunct)

The BT ("bank times") command formerly displayed resource allocation and usage information from the Resource Allocation and Control database for the current shift within the DPCS system. Allocations are now shown by using PSHARE (see the Priority (page 60) section below, or see the EZJOBCONTROL (URL: http://www.llnl.gov/LCdocs/ezjob) guide). Retrospective time usage is now reported by running PCSUSAGE, as described in the Bank and Allocation Manual (URL: http://www.llnl.gov/LCdocs/banks).

RA (Defunct)

The RA command formerly displayed resource allocation information by shift (day, night, weekend) from the Resource Allocation and Control database. Shifts are no longer used for (share) allocations or time reporting.

You can now report actual time used by whole day (0:00 to 24:00 only) by running the PCSUSAGE tool, as described and illustrated in the Bank and Allocation Manual (URL: http://www.llnl.gov/LCdocs/banks).

NEWACCT (Set Current Account)

NEWACCT is the DPCS account-assignment utility. It is used to change or set a user's current account (strictly voluntary now). Typical examples of using NEWACCT include:

```
newacct
The utility enters interactive mode.  It shows you the available
accounts  you  may charge to and prompts you for the account you
wish to be made your current account.  You may leave this  field
blank, and no change will occur.

newacct 590001
This command will cause your current account to be set to 590001.

newacct -l
This command will show you the current account for the
session.
```

DEFACCT (Set Default Account)

The DEFACCT utility is used to change or set a user's default account (strictly voluntary now). Typical examples of using DEFACCT include:

```
defacct

     The utility enters interactive mode.  It shows you the available
     accounts  you  may charge to and prompts you for the account you
     wish to be made your default account.  You may leave this  field
     blank,  in which case your default account is not changed.

defacct 590001

     This command will cause your default account to be set to 590001.

defacct -l bwood

     This command will show you the default account and bank for user
     bwood.
```

RACMGR (RAC Manager Daemon)

RACMGR runs on the control host. RACMGR is the principle daemon in the RAC subsystem. It monitors resource usage by sessions within the system, suspends or kills sessions as needed, and reports the resource utilization to the accounting system.

RACMGR is a sibling of RACRPT, RACCOM, and PWSD (all are control-host daemons). When RACMGR begins executing, its pipes to other processes have been set up already by its parent process pcsstart (which becomes RACCOM). RACMGR accomplishes its goals on each DPCS production machine by means of three other daemons, each of which runs on each production machine and reports to RACMGR on the control host machine:

RACCTD — makes a TCP/IP connection to both the ACCTD and the RACMGR daemons. It passes messages between these two daemons. The purpose of RACCTD is to provide a machine-independent interface between the ACCTD (which only knows the machine it is on) and the RACMGR (which knows little about the machine the ACCTD is on).

ACCTD — is a machine-dependent accounting daemon. Each production host may have different facilities for collecting the desired accounting information, so each production host must provide an accounting daemon to mediate between the local machine and the DPCS system. ACCTD collects information from the local system, and passes the data on to RACCTD. RACCTD reformats the information into the form expected by the RAC system, and passes the data to RACMGR.

RACSLV — is the resource allocation slave daemon. An instance of RACSLV runs on each production host. RACSLV performs auxiliary tasks for the RACMGR. The RACMGR may be controlling allocations on a remote machine, so RACSLV does actual control functions on each controlled machine.

In addition to these three production-machine daemons, RACMGR interacts with another helper daemon that runs on the control machine:

RACRPT — is the DPCS accounting report daemon. RACRPT receives records from RACMGR, reformats them, and sends them on to an accounting system or a file. When RACRPT begins executing, it has one input pipe open from RACMGR. Session resource usage records arrive over this pipe. RACRPT sends these records to an accounting system, if installed, or simply to a binary file if not.

Production Workload Scheduler (PWS)

Users submit batch requests directly to the PWS for secondary submittal to the batch system, specifying (or defaulting to) the bank and account to be charged. One important function of the PWS is to keep the machine busy without overloading it. PWS does not schedule interactive work, but it does track the resulting resource load and adjust the amount of production to "load level" the machine accordingly.

The set of production requests managed by the PWS is called the production workload. Requests (jobs) in this workload are prioritized according to rules and allocations laid out by system administrators and coordinators. High priority requests are permitted to run if the machine is not overloaded.

The PWS offers users a utility to submit batch jobs (PSUB) and 6 other utilities to manipulate the scheduling of those jobs. Several other software daemons interact with the utilities to manage the jobs submitted. An earlier figure (page 8) shows how these daemons and utilities are interrelated, while the following subsections describe them in greater detail.

The node in a multinode system where the DPCS daemons run (formerly called many things, including CWS and PRODHOST) is now called the "DPCS gateway node." On IBM SPs, the gateway node no longer needs to be the control workstation (CWS). LRMMGR assigns this gateway node.

When DPCS enters "installing mode" (for system updates), communication with DPCS daemons is disabled. Users receive a message that an installation is underway and that they should retry user utilities later.

The PWS Daemons (PWSD, PLSD, BCD)

The DPCS/LCRM Production Workload Scheduler involves three software daemons:

PWSD — is the Production Workload Scheduler Daemon. PWSD is the daemon that manages production for the DPCS. It is started by pcsstart and is a sibling process with other DPCS control daemons (running on the control host). It has unnamed pipes established between itself and RACMGR when it starts execution. This daemon now supports unlimited process table size, up to 300 hosts within a single DPCS/LCRM domain, and public/private RSA key authentication for security.

PLSD — is the PWS Load Statistics Deamon. PLSD reports load statistics on an AIX or Compaq UNIX machine. It is started by pcsstart and is an independent daemon that runs on each DPCS production machine.

BCD is the Batch Control Daemon. BCD is the daemon that manages the actual (underlying) batch system on each PCS-controlled (production) host. This isolates PWS functions from the particulars of any one batch system, allowing other parts of the PCS to control batch systems generically.

BCD executes as a server at a well-known, privileged port. The user of BCD must be a privileged client. The client contacts BCD using tcp/ip calls. It then sends an ascii string of tokens that represents the function desired to be executed. (See bcd_msg2a().) BCD parses this string (see a2bcd_msg()) and performs the function requested. It then returns an ascii string to the caller that represents the result of performing the function. BCD "hangs up" on the client after performing each function. BCD may get a request while it is processing another request. In this case, the new request is queued until the requests ahead of it are completed. Queued requests are processed FIFO. There is an API that should be used to perform BCD functions because it implements (and hides) the communication protocol for the user. It is libbcd.a. This API is not available to normal user processes.

BCD implements the following functions:

```
bcd_submit: Registers a batch job with a bcd.

bcd_move:   Causes a batch job to be moved from its host of
            submission to the host on which it will run.

bcd_run:    Causes the bcd to request its batch system to run the
            job.

bcd_hold:   If the job is running and the batch system does not
            support checkpointing, the response to this request
            is an error status.  Otherwise, if the job is running,
            it is checkpointed.  Otherwise, this function is a
            "noop".

bcd_kill:   Notifies that a job is to be removed regardless of its
            status.  If the job is running, it is killed.  If this
            host is the job's current home and it is not running,
            it is deleted from the batch system as well.

bcd_stat:   Returns to the caller the state of all jobs known to
            DPCS.
```

The PWS User Utilities

Users interact with DPCS-managed batch systems by running any of seven utilities that submit jobs or manipulate submitted jobs. The list below reveals the names of these utilities and the basic role of each; for practical advice on how to use them consult the comparisons and examples in the EZJOBCONTROL (URL: http://www.llnl.gov/LCdocs/ezjob) guide. Someday perhaps a detailed analysis of the options for each utility will appear here.

PSUB — submits your specified script to the batch system to run, with the time, memory, and other constraints that you indicate using PSUB options. For usage advice, traps, and examples, see the relevant section (URL: http://www.llnl.gov/LCdocs/ezjob/index.jsp?show=s6.5) of EZJOBCONTROL. Authorized users can also expedite jobs, exempt jobs, and force job priorities by using privileged features of PSUB (see below (page 47)).

PALTER — changes specified features of your already submitted batch job(s). Not all features can be altered. For usage advice, traps, and examples, see the relevant section (URL: http://www.llnl.gov/LCdocs/ezjob/index.jsp?show=s6.7) of EZJOBCONTROL. Authorized users can also expedite jobs, exempt jobs, and force job priorities by using privileged features of PALTER (see below (page 47)).

PEXP — allows authorized users to "expedite" a batch job so that it competes favorably against jobs funded from other PCS banks. Expanded PSUB and PALTER features have made the use of PEXP obsolete (see below (page 48)), although it persists for historical continuity. For usage advice, traps, and examples, see the relevant section (URL: http://www.llnl.gov/LCdocs/ezjob/index.jsp?show=s6.5) of EZJOBCONTROL.

PHOLD — makes a specified, submitted batch job ineligible to run until you release it with PREL. PALTER (page 51) can now achieve the same effect in another way. For usage advice, traps, and examples, see the relevant section (URL: http://www.llnl.gov/LCdocs/ezjob/index.jsp?show=s6.7) of EZJOBCONTROL.

PLIM — reports seven seldom-changed system default limits that your batch job faces on the machine where you run PLIM (such as maximum allowed run time and node-hour limits). For usage advice, traps, and examples, see the relevant section (URL: http://www.llnl.gov/LCdocs/ezjob/index.jsp?show=s6.4.1) of EZJOBCONTROL. For a system-specific configuration summary on each production machine, also consult the text file called /usr/local/docs/job.limits (where details vary by host).

PREL — releases a specified batch job to compete to run normally, after you have previously used PHOLD to hold it. For usage advice, traps, and examples, see the relevant section (URL: http://www.llnl.gov/LCdocs/ezjob/index.jsp?show=s6.7) of EZJOBCONTROL.

PRM — removes from the batch system a specified job that you had previously sumbitted, including jobs that have started to run. For usage advice, traps, and examples, see the relevant section (URL: http://www.llnl.gov/LCdocs/ezjob/index.jsp?show=s6.8) of EZJOBCONTROL.

DPCS Operating Features

This section discusses some of the inner machinery and hidden algorithms of DPCS, and tries to explain how they cause the reports, states, and overt behavior that job-running users may encounter.

Status Values for Batch Jobs

Interpreting Status Values

PSTAT's Role.
You can discover your batch job's unique DPCS identifier (its JID), monitor the job's status, and remind yourself of its (alterable) attributes by running the PSTAT utility. For PSTAT usage advice, traps, and examples, see the relevant section (URL: http://www.llnl.gov/LCdocs/ezjob/index.jsp?show=s6.6) of the EZJOBCONTROL guide.

Every 20 seconds DPCS evaluates submitted jobs to determine which, if any, should be scheduled. Nonscheduled jobs are assigned one of many possible states, supposed to reveal to users why the job is not running. Other states indicate that a job is running, or has stopped for some reason after starting to run. (In PSTAT output, an asterisk (*) precedes every job state for a job that has not yet run.)

Interpretation WARNINGS.
The order in which DPCS checks state conditions for scheduling appears in the job-scheduling section (page 37) below. The explanation of each state that DPCS can assign and PSTAT can report appears in the alphabetical list in the next subsection. Some of these status values (e.g., ELIG) can be ambiguous or misleading without careful interpretation.

Implicit in many DPCS status-value explanations is the concept that if a job could run on any of several machines, it has a (perhaps DIFFERENT) status associated with each separate machine. When you submit a batch job using PSUB, DPCS builds a list of permitted hosts for it. Your PSUB "constraints" (specified with the -c option) can overtly restrict this host list. But in general, every active DPCS-controlled machine (every production IBM machine, or Linux or Compaq cluster) goes into a job's permitted-host list.

If you query "the status" of a job with PSTAT before the job starts to run on some specific machine, the result may be ambiguous. Formerly by default PSTAT reported a job status for whatever host happened to be FIRST in the job's permitted-host list. However, depending on how each machine is configured, this default first-host status might not apply to other machines: a job could exceed your jobs-per-user quota (QTOTLIMU) on the first host, be TOOLONG to run on the second host, yet be ELIG or WMEM on the third, etc. Only by explicitly polling your job's status on each separate machine where it could run, using PSTAT's -m *hostname* option, for example,

```
pstat -m gps320 -u yourname
```

could get you an unambiguous report on why it had not started to run on each candidate host.

Now, however, by default PSTAT reports MULTIPLE as your job's status if it could run on any of several clustered machines with perhaps a different status on each. You can either still use -m *hostname* as above to disambiguate these incomplete status reports, or you can use PSTAT's -M (uppercase em) option, as shown here:

```
pstat -M -n 1234
```

-M reports a separate line with a separate status for each possible target machine where your job (specified by job-id with -n) could run. You cannot use both -M and -m on the same PSTAT execute line.

Alphabetical List of Status Values

(In PSTAT output, an asterisk (*) precedes every job state for a job that has not yet run.)

ACCOVER — The account being charged by this job has a quota and that quota has been exceeded.

BAT_WAIT — The job has been scheduled to run by DPCS but the underlying batch system on the production machine (NQS or LoadLeveler) has confirmed that the batch system itself is waiting for some resource before starting the job.

CMPLETED — (Shows only if you use PSTAT's open-side -T option to report on done jobs.) The job has finished running on its own (without being externally stopped), though not necessarily with success.

CPU&TIME — The product of CPUs requested and time requested for this job exceeds the maximum allowed on the target machine. [Exemptable. (page 50)]

CPUS>MAX — The job requires more dedicated CPUs on a machine than the machine's administrator will permit (per job) at the time of the status report. If the maximum allowed CPUs per job is later increased, the job will be reevaluated for scheduling. [Exemptable. (page 50)]

DEFERRED — LoadLeveler (on an IBM SP machine) has erroneously reported that no job classes exist, so this job has moved into DEFERRED state to wait 10 minutes to let the scheduler try again to get an appropriate LoadLeveler response and run the job (the retry delay is administratively configurable).

DELAYED — The job exceeds the current maximum number of allowed jobs (page 37) per user that DPCS/LCRM will actively consider for scheduling, but it falls below the maximum number of allowed delayed jobs. As other jobs are scheduled, delayed jobs automatically move (first in, first out) into active consideration. Attempts to submit more jobs than the allowed maximum number of delayed jobs are rejected outright.

DEPEND — The job is awaiting completion of another specific job on which it depends.

ELIG — The job is eligible to run. Currently, being "eligible" is a four-way ambiguous condition, since newly submitted jobs are assigned the ELIG state if
(1) no hard condition would prevent them from running (such as overt dependency on another, unfinished job), or
(2) they have been evaluated to run but another job is already scheduled to run, or
(3) scheduling evaluation reveals another job with a higher priority that itself cannot be scheduled to run yet, or
(4) scheduling this job would cause the load on its target machine to exceed a threshold set by the machine's administrators.
Plans call for assigning four distinct states to these four conditions at some future time.

ELIG_SBY — The job is eligible to run, but only at standby.

HELD*n* — The job has been explicitly held (using PHOLD) by either the user (U) who submitted it, the user's coordinator (C), the PCS system manager (P), or some combination of these three. Users can release their own holds, coodinators can release user holds as well as their own, and PCS managers can release all holds (by using PREL).

The hold level *n* reveals who has placed the hold(s) on each job, according to this chart:

Who has placed hold(s)			Hold
U	C	P	Level
X			1
	X		2
X	X		3
		X	4
X		X	5
	X	X	6
X	X	X	7

HLD_IDLE — A user-level hold has been applied to the job because its use of CPU time has fallen below a minimum threshold. The submitting user can remove this hold by running PREL.

HOLDING — The job is in the process of being removed from the run queue to be checkpointed (only applies to machines that support checkpointing). After the job is checkpointed, its status changes to HELD*n*, ELIG, or something else, depending on the reason for the checkpoint.

JRESLIM — The job exceeds the maximum number of concurrent jobs per bank or per user allowed in this whole resource partition. See Resource Partition Limits (page 30). [Exemptable. (page 50)]

MULTIPLE The job has not yet started to run, and so it has a separate (perhaps unique) status associated with each one of the multiple machines on which it might run later (e.g., each machine in a cluster). To report the job's unambiguous status for one specific machine, use PSTAT's -m *hostname* option. To get a multiline report showing the job's status on every machine where it could later run, use -M -n *jid*.

NEW The job has not yet been evaluated by the production workload scheduler (precedes ELIG and other postevaluation states).

NOACCT The account to be charged for this job's resources (or the job owner's permission to charge that account) has been removed.

NOBANK The bank from which the job was to draw its resources no longer exists, or the submitting user no longer has permission to charge against that bank.

NOCONF A machine where the user permitted the job to run has no valid configuration parameter set assigned (this is an administrative error). The job's NOCONF status for that machine prevents scheduling it on that machine, but it may still be scheduled on another machine if permitted by the submitting user.

NONEW An administrator has instructed DPCS to stop scheduling new jobs on a machine where the submitting user has permitted the job to run. The job's NONEW status for that machine prevents scheduling it there, but it may still be scheduled on another machine if permitted by the user.

NOPRISRV The machine that the job is selected to run on is operating at a priority service level (greater than normal), and the job's bank is not within the priority-service bank (sub)tree. This status will persist until the machine returns to a normal priority service level, or until the job is scheduled on a different machine without this constraint.

NOTIME The amount of time that will likely be consumed by the job exceeds the user's remaining time in the bank from which the job is drawing resources on a machine where the user permitted the job to run. The job's NOTIME status prevents scheduling the job on that machine, but it may still be scheduled on another machine without this constaint. However, jobs that linger in NOTIME status for a "prolonged period" are purged to simplify future scheduling decisions.

NRESLIM The job exceeds the maximum number of nodes per bank or per user allowed in this whole resource partition. See Resource Partition Limits (page 30). [Exemptable. (page 50)]

NTRESLIM The job exceeds the maximum amount of node time per bank or per user allowed in this whole resource partition. See Resource Partition Limits (page 30). [Exemptable. (page 50)]

PREEMPTD — This formerly running job has temporarily stopped execution (but it remains memory resident) to let an expedited (page 48) job use its nodes. PREEMPTD jobs charge no time while they wait and automatically resume execution when the expedited job ends. (IBM SP machines only.)

PTOOBIG — The maximum process size of the job exceeds the maximum permitted size of processes on a machine where the user permitted the job to run. This prevents scheduling the job on that machine, but it may still be scheduled on another machine that allows larger processes.

QCKPLIM — The amount of available checkpoint space on a machine where the user has permitted the job to run is less than a preconfigured minimum limit. This prevents the job from being scheduled on that machine, but it may still be scheduled on another machine if permitted by the user. Also, if more checkpoint space becomes available, DPCS will reevaluate this status.

QTOTLIM — The total number of batch jobs currently running on a machine where the user has permitted the job to run matches or exceeds the maximum number of running jobs allowed (by an administrator). This prevents scheduling the job on that machine, but it may still be scheduled on another machine if permitted by the user. [Exemptable. (page 50)]

QTOTLIMU — The total number of batch jobs owned by the user currently running on a machine where the user has permitted the job to run matches or exceeds the maximum number of running jobs allowed for any one user (by an administrator). This prevents scheduling the job on that machine, but it may still be scheduled on another machine if permitted by the user. [Exemptable. (page 50)]

REMOVED — (Shows only if you use PSTAT's open-side -T option to report on done jobs.) The job has been removed from the batch system by someone (its owner or a manager) running PRM.

RES_WAIT — The job requires more units of a declared nonshareable resource than are currently available on its target machine(s). As the Managing Nonshareable Resources (page 46) section explains, DPCS no longer supports nonshareable resources so this status has become obsolete.

RM_INIT — The job is in the process of being removed from the system (by running PRM).

RM_PEND — The job is in the process of being removed but the removal request is not yet completed by the production machine where the job is running (DPCS is awaiting removal confirmation by the production machine's daemon).

RUN — The job has been scheduled to run by DPCS and the underlying batch system (NQS or LoadLeveler) has confirmed that it is running.

RUN_SBY The job has been scheduled to run by DPCS and the underlying batch system (NQS or LoadLeveler) has confirmed that it is running, but only at standby (subject to a warning signal or, if not registered for a signal, to immediate termination).

STAGING The job has been scheduled to run by DPCS but the underlying batch system (NQS or LoadLeveler) has not yet confirmed that it is running.

TERMINATED

(Shows only if you use PSTAT's open-side -T option to report on done jobs.) The job was killed either by a DPCS manager or because it ran out of time.

TOOLONG The job's requested time limit exceeds the maximum amount of time allowed for jobs on a machine where the user permitted the job to run. This prevents the job from being scheduled on that machine, but it may still be scheduled on another machine that allows longer jobs. Note also that requesting more time than a machine allows does NOT remove that machine from the job's permitted-host list (page 19). So a job may be reported as TOOLONG for several machines yet simultaneously be eligible to run on several others (use PSTAT's -m option to check each machine separately). [Exemptable. (page 50)]

TQUOTA On a machine where the user permitted the job to run, the user's allocation or bank has a per/user resource quota and the user has reached that quota. This prevents the job from being scheduled on that machine, but it may still be scheduled on another machine if the user permits.

WAIT The user has specified the earliest time that the job is permitted to run, and that time has not yet arrived.

WCPU Insufficient nodes are available to allow running this job (with its requested node count) at this time.

WHOST DPCS's production workload scheduler daemon (pwsd) is not connected to the PCS daemon on a machine where the user has permitted the job to *run*. DPCS assigns WHOST as the job's status for the unconnected machine, but the job may still be scheduled on another machine if the user permits (it may eventually be scheuduled for the original machine if the system administrators correct the problem). WHOST may also simply indicate that a specific machine's administrators have instructed DPCS not to run any jobs on that machine (temporarily). See also WSUBH.

WMEM The machine(s) on which a job is permitted to run are already loaded to the extent that scheduling this job would overload memory.

WMEML The load on the machine(s) on which the job is permitted to run is already higher than the maximum load desired by the machine administrators (overloaded). [Exemptable. (page 50)]

WMEMT	The load on the machine(s) on which the job is permitted to run is already as high as the maximum load desired by the machine administrators (properly loaded).
WPRIO	This job is not scheduled to run because scheduling it would delay execution of another job with a higher priority.
WSUBH	DPCS's production workload scheduler daemon (pwsd) is not connected to the spooler daemon on the machine where the job was *submitted*. See also WHOST, which formerly covered this conditon as well.

Class Values for Batch Jobs

DPCS recognizes five job classes, which PSTAT usually reports using the following single-letter codes:

N — indicates a normal job. This is the default job class and most batch jobs are class N.

P — formerly indicated a "short-production" job. Authorized users (only) could put a job in the short-production class by using PSUB's former -sp option. In January, 2003, DPCS stopped supporting -sp and all short-production jobs.

S — indicates a "standby" job. Standby jobs increase machine utilization by taping cycles that would otherwise remain idle. A standby job has such a low scheduling priority that it runs only when no normal or expedited jobs are available to run (on a target machine) and only if scheduling it will not slow the throughput of other normal or expedited jobs already running. Furthermore, DPCS will terminate a standby job to make its nodes or memory available if needed for any normal or expedited job that becomes eligible to run after the standby job has started. If the standby job has registered to take a warning signal (page 64), DPCS will signal it and allow the grace period (configured for that machine) before termination. Otherwise, the job terminates at once (no unsignalled grace period). Standby jobs are terminated "abnormally," never preempted so they can resume later. Use the PSUB or PALTER -standby option to request this job class (see EZJOBCONTROL (URL: http://www.llnl.gov/LCdocs/ezjob)).

X — indicates an "expedited" job. (page 48) Authorized users only can put a job in the expedited class by using the PEXP (or special options of the PSUB or PALTER) utilities.

indicates a "nonstop" or "nonpreemptable" job. On IBM SP machines only, PSUB's -np option can place a job in a special class that prevents it from being preempted for gang scheduling for up to 2 hours.

Sometimes LoadLeveler (on an IBM SP machine) erroneously tells the DPCS job scheduler that no classes exist. Pending jobs then move into the DEFERRED state to wait 10 minutes so the scheduler can try again to get a more appropriate LoadLeveler response.

Beginning in 2001, DPCS supports the ASCI tri-lab policy of allowing jobs in different classes (with different levels of service) to accumulate charges against their owner's fair-share allocation at *different* rates.

Run Properties of Batch Jobs

PSTAT optionally reports on many relevant properties of running (or recently completed) batch jobs (besides their status (page 19) and class (page 26) values, described above). Using

pstat -n *jid* -f

"fully" reports the run properties of job *jid*, and the (nonobvious) fields in this full report are explained below. Using

pstat -n *jid* -o *prop1,prop2,...*

reports just on the specific properties *prop1* etc. that you specify, using as literal strings the field names listed in the explanatory guide below.

The available run properties on which PSTAT optionally reports are listed here in alphabetical order by PSTAT's -o field name (the corresponding descriptive label for -f reports appears in parentheses whenever it is significantly different):

CL — is the job's class (page 26), most useful for revealing if it has been successfully expedited (class X).

CONSTRAINT — shows the values that you specified with PSUB's CONSTRAINT (-c) and GEOMETRY (-g) options when you submitted this batch job (they usually limit the machines or nodes on which the job can run).

CPN — is CPUs per node, which SLURM manages as a separate, identifiable job constraint on Linux (CHAOS) systems.

EARLIEST_START — ("earliest start time") is the earliest date and time at which your job will begin to run (if optionally specified by you using PSUB's -A option when you submitted the job). Formerly called "do not run before" on PSTAT -f reports.

ECOMPTIME — ("estimated completion") is the date and time at which your job will most likely finish running. This estimate changes continuously because DPCS computes it using a heuristic algorithm involving the CPU per-task time limit, the elapsed run time limit (if any), the forced stop time (if any), and the rate at which the job is now using time (time used divided by elapsed run time). Of course DPCS cannot predict when jobs will abort because of internal flaws. For completed jobs, this field is instead reported as "terminated at."

HIGHWATER — ("largest process size") is the largest individual process size ever reached by this job (its memory "high-water mark" so far). See also MEMSIZE, MAXPHYSS, and MAXRSS for related values.

MAXCPUTIME

("time limit per task") is the maximum (average) per-task time limit that you declared when you submitted this job. When the average (not total) time used by all tasks in a job exceeds MAXCPUTIME, DPCS terminates the job.

MAXMEM ("process size limit") is the per-process memory size limit that you declared when you submitted this job.

MAXNODES is the same as NODES (see below).

MAXPHYSS ("maximum physical size") is the maximum virtual memory actually used by the job (per node) so far.

MAXRSS ("maximum resident set size") is the maximum real memory actually used by the job (per node) so far.

MAXRUNTIME

("elapsed run time limit") is the maximum wall-clock time for this job that you (optionally) declared with PSUB's -tW option when you submitted the job (reported in hours:minutes). RUNTIME (below) shows the wall-clock time used so far.

MAXTIME is the former name of MAXCPUTIME, retained only for backward compatability when you use PSTAT's -o option.

MEMINT ("resident memory integral") is the resident set memory integral in megabyte hours (see also VMEMINT below).

MEMSIZE ("job size") is the job's current total memory size (the last measured sum of the memory used by all processes in this job). See also HIGHWATER.

NODES ("node distribution") is the node count or range of nodes *requested* by a job that has not yet started to run, and it is the actual number of nodes *assigned* to the job after it has started to run.

RUNTIME ("elapsed run time") is the elapsed wall-clock time since this job began executing (reported in hours:minutes). The limit on RUNTIME (if any) is in MAXRUNTIME.

SUBMITTED ("submitted at") is the time of day and date at which this job was submitted to DPCS/LCRM by PSUB.

TASKS is the number of tasks for this job, which DPCS/LCRM calculates using the algorithm shown in this table (where task count depends both on platform and on run technique):

DPCS Task Count for Different Job Contexts

Platform:	Compaq (GPS, Tera, SC)	IBM SP (Blue, SKY, White)
Run Technique:		
Uses geometry option -g		tasks = number of requested nodes times number of requested tasks/node
Omits geometry option -g		tasks = number of requested nodes
Registers with gang scheduler	tasks = number of requested tasks	
Runs without gang scheduler	1 task only	

TIMECHARGED

("time charged") is the total CPU time used by all tasks in a job in hours:minutes (much larger than USED if the job has many tasks, but the same if it has only one task). See TASKS for how DPCS/LCRM computes your task count.

USED ("time used per task") is the average CPU time used per task in hours:minutes (not the total time used by all tasks, which is TIMECHARGED).

VMEMINT is the "physical memory integral" in megabyte hours (see also MEMINT above).

XCT ("expedited count") has been replaced by CL (above).

Resource Partition Limits

In February, 2002, DPCS began supporting "resource partition limits" for each bank and each user. A DPCS "resource partition" is a set of similar or related computers (such as all GPS nodes) that DPCS manages together when scheduling jobs. (See the Bank and Allocation Manual (URL: http://www.llnl.gov/LCdocs/banks) for the current list of DPCS resource partitions.)

For each separate resource partition, DPCS administrators can set a limit on:

- the maximum number of concurrent jobs allowed per bank, per user, or both,
- the maximum number of nodes committed to (jobs from) each bank, each user, or both, and
- the maximun node time (in minutes) allowed for (jobs from) each bank, each user, or both.

(The default initial setting for all three limits is "unlimited.")

EFFECTS.
(1) Enabling these partition limits will prevent some batch jobs from running in a "global" way that may not be apparent by looking at each job's characteristics alone. For example, if the maximum number of concurrent jobs allowed per bank (in a partition) is four, then DPCS will not schedule any fifth job that draws resouces from that bank to run, regardless of how the four running jobs are spread among the many nodes or users in that partition.
(2) Once any enabled partition limit is hit, DPCS will assign to all queued and not yet running jobs a new status (page 19) that reveals which limit prevents the job from running:

JRESLIM	means that the job would exceed the maximum job limit.
NRESLIM	means that the job would exceed the maximum node limit.
NTRESLIM	means that the job would exceed the maximum node-time limit.

(3) Enabling these limits will reduce machine utilization and may cause scheduling anomalies (such as not running low-priority jobs as "backfill" for high-priority jobs). If the limits are set on high-level banks, there is no easy way to identify just which jobs (that charge against the children of these banks) are causing queued batch jobs not to run. For predictable scheduling, user limits may prove easier to use than bank limits.

REPORTING.
The BRLIM utility reports for a specified bank or user (or set) the currently assigned values of all three possible "resource partition" limits and the current commitment of resources (called "usage") against each limit. This helps predict whether a proposed new batch job would exceed any relevant limit and thus not be scheduled. Note that BRLIM does *not* report on individual jobs (as PSTAT does), but rather on user or bank aggregate limit commitments (very like traditional allocations). See the BRLIM section of LC's Bank and Allocation Manual (URL: http://www.llnl.gov/LCdocs/banks) for execution details, control options, and annotated usage examples.

EXEMPTIONS.

DPCS resource partition limits are "exemptable." Authorized users (page 52) (only) can invoke the -exempt option of PSUB or PALTER to override limits that would otherwise prevent their batch job from running. (Actually, the limit status values JRESLIM, NRESLIM, and NTRESLIM are used to request the exemption.) See the Exempting Jobs (page 50) section below for full instructions on how to use such exemptions.

LOCAL LIMITS.

One may easily confuse these three global job limits, that apply to an entire DPCS resource partition, with the more familiar local limits that apply to specific nodes in each cluster of machines (for example, some GPS nodes allow more jobs/user than others, and some LX nodes allow much more time/job than others). Such local limits are *not* reported by BRLIM; instead consult LC's composite job-limits web page (uses OTP authentication) at

`https://lc.llnl.gov/computing/status/limits.html`

The BRLIM-reported JRESLIM, NRESLIM, and NTRESLIM limits (if not set to "unlimited") are superimposed over the local limits, which is why predicting their effect on your job is so tricky.

Environment Variables for Batch Jobs

On machines (such as Compaqs) where NQS is the low-level batch system underlying DPCS, the environment variables related to your batch job are handled as follows:

SAVED AUTOMATICALLY.

At the time you submit the job, six of your current environment variables are saved (exported) and then automatically restored with new names for use while the job executes. They are:

Original Environment Variable	Automatically Saved/exported As
HOME	QSUB_HOME
LOGNAME	QSUB_LOGNAME
MAIL	QSUB_MAIL
PATH	QSUB_PATH
SHELL	QSUB_SHELL
TZ	QSUB_TZ

SAVED ONLY BY REQUEST.

All your other original environment variables are NOT saved (exported) by default. However, at the time you submit the job you can overtly request that DPCS save (export) all of your environment variables by using PSUB's -x option. Running PSUB with -x saves every environment variable (other than the six saved automatically above) under the same name they had originally, without prepending QSUB_ to the name.

CREATED AUTOMATICALLY.

In addition to the two groups mentioned already, NQS automatically creates four more environment variables and associates them with your batch job. These extra variables and their assigned values are:

QSUB_HOST contains the name of the host (machine) where your job originated.

QSUB_REQID

contains the NQS identifier assigned to your job ("request").

QSUB_REQNAME

contains the NQS name of your job ("request"), as you specified by using PSUB's -r option.

QSUB_WORKDIR

contains the pathname of the current work directory at the time you submitted your job.

You can use any of the above environment variables within your batch script, to help your job run as you intend, as long as you take account of the value they will have according to the rules explained here.

On machines without NQS as the underlying batch system (such as the IBM SP, which uses LoadLeveler instead), environment variables are handled differently. None of the ten QSUB-named environment variables mentioned above exist. They are therefore not available for use in your scripts.

To compensate, DPCS now supports a parallel but differently named set of environment variables that are available on ALL machines and that fill the roles the NQS-provided variables would have filled:

PCS_TMPDIR

specifies the location of a temporary working directory that DPCS creates when a job starts, that persists during the whole job, and that is automatically purged when the job completes. System administrators toggle the use of this environment variable and configure the directory name, if any (if none, then DPCS creates no such temporary directory). See also PSUB_SUBDIR and PSUB_WORKDIR below.

PSUB_HOME

preserves the HOME environment variable in effect when you submitted your job.

PSUB_HOST contains the name of the host from which the job was submitted.

PSUB_JOBID contains the identifier assigned to the job by DPCS.

Echoing the value of this PSUB_JOBID variable in your job script can avoid a common job-tracking problem with DPCS. The PSUB job-monitoring utility reports only on "waiting" or running jobs, not on completed jobs. So you will not be able to discover the job ID of a completed job once it ends, a debugging obstacle if you run several jobs. It is therefore good practice to include a request to echo into your log file the value of this environment variable at the start of every job, thusly:

```
echo DPCS job id = $PSUB_JOBID
```

The output will be the 5-digit number that uniquely identified the current job to DPCS while it ran.

PSUB_LOGNAME

preserves the LOGNAME environment variable in effect when you submitted your job.

PSUB_REQNAME

contains your job's request name, that you specified with PSUB's -r option.

PSUB_SHELL

preserves the SHELL environment variable in effect when you submitted your job.

PSUB_SUBDIR

contains the pathname of the current directory in effect on the machine from which you *submitted* your job. See also PSUB_WORKDIR.

PSUB_TZ preserves the TZ (time zone) environment variable in effect when you submitted your job.

PSUB_USER preserves the LOGNAME environment variable in effect when you submitted your job, the same as PSUB_LOGNAME.

PSUB_WORKDIR

(formerly indentical to PSUB_SUBDIR, changed January, 2003) contains the same value as PCS_TMPDIR if that environment variable is set. Otherwise, contains the pathname of your home directory on the *execution* (not the submittal) machine. See also PSUB_SUBDIR.

You can use these PSUB-named environment variables in any script that will run under DPCS, and using them instead of the NQS-based, QSUB-named variables will give your script independence from NQS. Scripts using the PSUB-named variables can run on IBM SP and Linux machines as well as on machines from Compaq.

Massively parallel batch jobs usually depend on additional, specialized environment variables (some available only for noninteractive jobs) that control the behavior of (1) the message-passing interface (MPI) for between-process communication, and (2) any POSIX threads (pthreads) that the job may create for within-process concurrency. For the roles of these "parallelization" environment variables and their default values under AIX on LC's IBM machines, see the POE (Parallel Operating Environment) User Guide. (URL: http://www.llnl.gov/LCdocs/poe)

On LC Linux (CHAOS) machines, where SLURM is the low-level batch system underlying DPCS/LCRM, SLURM uses its own distinct set of environment variables to support each executing task (all begin with the string "SLURM_"). See the "Environment Variables" section of the SLURM Reference Manual (URL: http://www.llnl.gov/LCdocs/slurm) for an explanatory list of SLURM's unique variables.

Comment and Shell Handling

The basic and typical uses of comments within batch scripts are shown and explained in the "Annotated Typical Batch Script" section (URL: http://www.llnl.gov/LCdocs/ezjob/index.jsp?show=s4) of the EZJOBCONTROL guide.

The underlying rules for batch-script comments under DPCS are:

- Using # in the first column makes a line a comment. For example,

```
#this is a comment line.
```

- Using the syntax #!*shellpath* on the first line of your script makes a special comment that declares the shell your job should invoke. For example,

```
#!/bin/csh
```

  invokes the C shell.

- Using #PSUB (all uppercase) makes a comment line into an imbedded command to the PSUB job-submittal utility. Imbedded PSUB commands have the same effect as options on PSUB's interactive execute line. For example,

```
#PSUB -r jobname
```

  declares a nondefault name for your job ("request").

- Explanatory comments can be included at the end of the same line with imbedded PSUB commands by preceding the comment string with #, such as

```
#PSUB -r jobname # declares job's name
```

- Scripts inherited from native-NQS batch systems that use QSUB commands may end up with imbedded PSUB and QSUB commands mixed as comments at the start of the script, such as:

```
#!/bin/csh
#PSUB -r job1
#QSUB -r job2
#other comments here
```

  This is harmless because of the special way DPCS handles comments, as explained below.

Because of possible PSUB/QSUB command mixing, DPCS actually REMOVES from all submitted scripts all comment lines other than the initial #!*shellpath* comment (if any) before forwarding the job file to the underlying batch system (e.g., NQS) to run. This "comment cleaning" eliminates any stray QSUB commands in the script that might otherwise contradict the PSUB commands that are intended to dictate (via DPCS) how the job executes. If you neglect to supply a shell-specifying #! comment, however, this process leaves no comment lines at all.

NQS (the underlying batch system on LC Compaqs) and SLURM (the underlying system on LC Linux machines) always set the default shell to the Bourne shell (SH), instead of to your login shell, if your script does not begin with a comment line. This default means that jobs intended to run under the CSH or Korn

shells could fail with serious errors. To compensate, the PSUB utility now guarantees that, despite the "comment cleaning" process just noted, at least one blank comment line remains at the top of every submitted script file. However, prudence suggests that you always specify your desired job shell overtly with a line of the form #!*shellpath* as the first line in every batch script you submit. This precaution completely avoids the danger that DPCS comment removal will cause your script to fail.

Job Scheduling

This section explains the DPCS/LCRM limitations on the number of jobs that any single user can submit, then gives the order in which DPCS/LCRM checks conditions that preclude scheduling a submitted batch job, then descibes in detail the job-scheduling algorithm itself.

Order of Checking Precluding Conditions

DELAYS.

DPCS/LCRM allows system administrators to specify (to configure by using the LRMMGR utility) the maximum number of jobs per user that it will actively consider for scheduling. The LRMMGR command

```
update global maxjobsperuser n
```

sets *n* as the maximum allowed jobs per user (integers greater than 0, up to the special value of UNLIMITED). Jobs that a user submits above this limit DPCS/LCRM delays from active consideration for scheduling. Such excess jobs get the PSTAT state DELAYED. As "active" jobs are gradually scheduled, DPCS/LCRM automatically moves (first in, first out) each user's delayed job(s) back into active consideration. (NOTE: if a system administrator increases MAXJOBSPERUSER while some jobs are already delayed, they will still gradually follow this first-in-first-out path, rather than all suddenly becoming active.)

DPCS/LCRM also lets system administrators specify the maximum number of *delayed* jobs per user. The LRMMGR command

```
update global delayedjoblimit n
```

sets *n* as the largest number of delayed jobs that any single user is allowed to accumulate (integers from 0 to UNLIMITED, inclusive). When a user reaches this delayed-job limit, DPCS/LCRM refuses to accept any more job submittals from that user. (NOTE: the LRMMGR command "show global" reports the current values of the two job limits discussed here, along with other global parameters.)

PRECLUDING CONDITIONS.

Every 20 seconds DPCS/LCRM evaluates submitted (and not delayed) jobs to see which, if any, should be scheduled to run. It begins by checking, in a specific order, a long list of conditions each of which precludes scheduling the job (and each of which corresponds to one job-status code that PSTAT can report).

An earlier section (page 19) explains and interprets these status codes (in alphabetical order for easy reference). Here we list the job-status codes in the order in which DPCS/LCRM checks their conditions to test if a job is precluded from scheduling:

Scheduling precluded because
job is already scheduled:
     STAGING
     BAT_WAIT
     RUN

Scheduling precluded
for another reason:
     DEFERRED
     DEPEND
     HELD*n*
     WAIT
     USED>MAX
     NOBANK
     NOACCT
     ACCOVER
     WHOST
     NOCONF
     TQUOTA
     NOPRISRV
     PTOOBIG
     TOOLONG
     NOTIME
     NONEW
     QTOTLIM
     QTOTLIMU
     QCKPLIM

Algorithm for Job Scheduling

If no conditions preclude a job from being scheduled on every available machine (see the previous section (page 37)), the job enters a pool of candidate jobs to which the DPCS scheduling algorithm is applied. More precisely, DPCS constructs a list of schedulable jobs for each machine in the DPCS domain. A job may be in several lists if the user has permitted the job to run on more than one machine. (Starting in 2001, DPCS does not attempt to schedule every available machine ("compute server") during every scheduling cycle. There are now so many different machines that the DPCS scheduling interval for each is configurable, to allow more flexible use of DPCS resources.)

JOB PRIORITY.
Each job is assigned a priority, and then each list of jobs is sorted by job priority. A job's overall priority for a machine, p[j,m], is a function, a weighted sum of three subpriorities:

```
p[j,m] = (tp[j] * tw[m]) + (ap[j] * aw[m]) + (pp[j] * pw[m])
```

where

tp[j] is the job's *technical* priority (a measure of its likely efficiency or ability to use resources well). DPCS looks at both the "memory advisory" hint that you (optionally) provide with the PSUB -lM option and at your (recent) historical memory usage patterns (which you can check by running PHIST) to estimate your job's memory demands when guessing its efficiency.

As of May, 2003, the official formula for technical priority is:

```
tp[j] =

[timeprioweight * MIN(1.0, requested_time/idealjobduration] +

[nodeprioweight *
MIN(1.0, 5.0/(5.0 + ABS(requested_nodes - idealnodecnt))]
```

In this formula, authorized users (page 52) (only) can use the LRMMGR (formerly PCSMGR) commands

```
create|update config cname aname avalue
```

to specify for a configuration *cname* a value *avalue* for any of the four technical-priority attributes *aname* shown above. In particular, *aname* can be:

timeprioweight (default is 0.0) specifies the fraction of the time-based term (first line above) used to compute the technical priority.

nodeprioweight (default is 1.0) specifies the fraction of the node-based term (second line above) used to compute the technical priority.

idealjobduration (default equals 24h) specifies the smallest job duration that would maximize the time-based term (first line above) used to compute the technical priority.

idealnodecnt (default equals 32 nodes) specifies the number of nodes that, if requested for a job, will maximize the node-based term (second line above) used to compute the technical priority.

ap[j] is the job's *aging* priority (a measure of its starvation for resources).

pp[j] is the job's *political* priority (a measure of the share of resources that have been consumed by the job's user or bank compared with the share of resources that should have been consumed).

tw[m] is the technical-priority *weight* for the target machine, set by its administrators.

aw[m] is the aging-priority *weight* for the target machine, set by its administrators.

pw[m] is the political-priority *weight* for the target machine, set by its administrators.

LOAD BALANCING.

The load on every DPCS-managed machine is sampled (there is a smoothing factor to damp oscillations). The list of machines is then sorted in inverse order by load (or from lightest loaded to heaviest loaded). Before August, 2001, DPCS considered only memory load, but now it considers both memory load and processor load on all shared SMP computers (to improve the performance of parallel jobs that demand many processors).

For each machine in order, DPCS tries to schedule one and only one job. If any job is scheduled on a machine, it will normally be the highest priority job in the list of jobs that can be scheduled on that machine. If a job is scheduled on a machine, the job is removed as a candidate to be scheduled on any other machines before those machines are evaluated to see if a job can be scheduled on them.

JOB SCHEDULING.

For each machine that DPCS manages:

- If the load on the machine exceeds target maximums, then, if the machine supports checkpointing, then the lowest priority running job is checkpointed. No job is scheduled on the machine.
- If there are no schedulable nonrunning jobs for the machine, then no job is scheduled on the machine.
- If the highest priority nonrunning job for the machine is an expedited or short-production (now obsolete) job, then it is scheduled on the machine.
- If the minimum number of high-priority jobs is not yet running on the machine and if scheduling the highest priority nonrunning job on the machine would not likely cause the load on the machine to exceed target maximums, then it is scheduled on the machine.
- If the minimum number of high-priority jobs is not yet running on the machine and if scheduling the highest priority nonrunning job WOULD likely cause the load on the machine to exceed target maximums, then if the machine supports checkpointing and if the highest priority nonrunning job has a priority higher than the lowest priority running job and if the lowest priority running job has already exceeded its do-not-disturb time, then it is checkpointed, but no new job is scheduled on the machine.
- If this point in the algorithm is reached, the minimum number of high-priority jobs is running on the machine. DPCS then picks the "best" job as a candidate to run on the machine. The "best" job is currently defined as the job with the highest *technical* (not overall) priority.
- If the "best" job would not likely cause the load on the machine to exceed its target maximums, then it is scheduled to run on the machine. (On Compaq machines, where RMS is the underlying batch system, DPCS will "backfill schedule," that is, it will start lower priority jobs as long as doing so will not delay the start of higher priority jobs.)
- If the load on the machine exceeds target minimums, and if either the machine does NOT support checkpointing or both the machine supports checkpointing and none of the running jobs have consumed their do-not-disturb time, then no job is scheduled on the machine.
- If the running job with the lowest priority is lower in priority than the "best" job that is not running, and if the machine supports checkpointing, then the lowest priority running job is checkpointed. No new job is scheduled on the machine.

- DPCS computes how over- or under-serviced a user or bank is by looking at both actual recent usage and the "anticipated cost" of currently running jobs, where the later is some fraction of each running job's requested time and nodes. DPCS managers can specify the fraction of running-job cost that they want used to compute the "anticipated cost" (for each separate resource partition) by using the LRMMGR input line

  `update partition` *`pname`* `comfact` *`costratio`*

  where *costratio* is a decimal number between 0.0 and 1.0 inclusive.

Output Truncation

DPCS truncates standard output from each executing program to 999,999 bytes. Therefore, if your batch job runs any programs that are likely to generate more than about 1 Mbyte of output, your script should explicitly redirect that output to a specific file instead of relying on standard output. A simple example would be

```
/usr/bin/spell test001 >! sp.out
```

Redirecting with file overwrite (>! instead of >) reduces the chance of failure if the job must be rerun in whole or in part because of a problem.

Reporting Memory and Time Used

Discovering for a specific job (rather than generally for a user or bank) the

- current memory size,
- high-water mark memory used, and
- computer time used so far

is often desirable. The PSTAT utility provides several options that report these three job features, either alone or as part of a general job summary.

Once a job (whose DPCS identifying number is *jid*) has started to run on a specific machine, you can use

```
pstat -n jid -f
```

to simultaneously report many job properties (a "full" report). Included in this summary are the job's current "job size" (last measured sum of the memory used by all processes in the job), the largest process size reached by the job (high-water mark so far), and total "time used" (in hours:minutes) by all tasks so far. Or you can use

```
pstat -n jid -o memsize,highwater,timecharged
```

(where "memsize" and "highwater" and "timecharged" are literal arguments to the -o option, NOT variables) to report exclusively on the interesting values of current and maximum memory size and total time used so far. (The literal "used" reports average time used per task, instead of total time for all tasks. Or consider the literals "maxrss" and "maxphyss" to separately report maximum real and virtual memory used per node.)

For 5 days after a job has completed you can still report its last-measured memory size, its high-water mark memory, and its total time used by typing

```
pstat -n jid -T -o memsize,highwater,timecharged
```

For information on done jobs later than 5 days after their completion, see the next section. For other fields that you can report with PSTAT's -o option, see the Run Properties (page 27) section above.

Log Files for Done Jobs

The log file that your own batch job makes for itself reveals the steps executed according to your batch script, but not the constraints or parameters with which you submitted the job nor the resource problems the job may have encountered. Sometimes after a job ends, especially if it died before successful completion, you may need to reconstruct exactly how you sumbitted it or why (or when) it got into trouble. Both parts of DPCS (RAC for resource allocation and PWS for scheduling) keep itemized log files that can sometimes answer these questions about done jobs.

For the first 5 days after a DPCS job ends, you can use PSTAT with the -T option (open machines only) to retrieve (some of) this log information yourself (see the previous section for tips).

Beyond the 5-day PSTAT limit, you can ask the LC Hotline staff (page ?) to go to the administrative machines HOCUS (open) or POCUS (SCF) and search the DPCS log files for you. On HOCUS or POCUS, log files reside in the directories

```
/usr/local/adm/pcs/pws.log
/usr/local/adm/pcs/rac.log
```

for one day and then they move to

```
/usr/local/adm/pcs/archive
```

for about three months. Running GREP in the appropriate directory to search for your job's unique identifing number (*jid*) will display log-file lines that reveal the job's characteristics when you submitted it and its interactions with DPCS job-management software as it started to run, hit a time (or other) limit, was deleted, and ended. Searching DPCS logs is an annoying process and the DPCS log files are poorly annotated, so you should rely PRIMARILY on your own good record keeping and timely use of PSTAT, not on these DPCS system logs, to debug resource problems with your batch jobs.

DFS and DCE Interactions with Batch

DFS is LC's Distributed File System, a separate set of disks managed by special software so that they appear as local disks on many physically distributed computers at once. DFS provides very fine-grained (user-by-user) control over access to individual files compare to ordinary UNIX (which can be important for export control purposes). And DFS provides a high level of security using DCE (Distributed Computing Environment) password management on machines where DCE is supported.

Before January, 2003, PSUB would try to get the DCE credentials of every user who submitted a batch job on any DCE-enabled machine. Now, LC's OCF machines use one-time passwords (OTP) instead of DCE passwords. And some massively parallel (IBM/POE) machines never did support DCE credentials and hence issued warnings for every submitted job.

Until January, 2003, users had to invoke a special -noDFS option when they ran PSUB to avoid these problems. Now, DPCS no longer supports DFS/DCE access in any way. Compensating precautions are no longer needed, and hence the former -noDFS option has disappeared from PSUB.

Managing Nonshareable Resources

Beginning in December, 1999, DPCS could be used to manage any computing resources declared to be nonshareable (such as local temporary disk space, software licenses, or tape drives). Special LRMMGR (nsresource), PSUB (-ns), and PSTAT (RES_WAIT) features were installed specifically to support nonshareable resource management. In January, 2003, all such features were removed from DPCS because no one used them.

Expediting and Exempting Jobs

Beginning in 2001 (on both OCF and SCF machines), the PSUB and PALTER utilities were enhanced to let authorized users independently:

- EXPEDITE a job (specify that it should start as soon as possible, even preempting other jobs to do so), or
- EXEMPT a job from specified system limits or administrative constraints on job size or number (also misleadingly called "statuses") that control when it normally runs, or
- force the execution PRIORITY of a job to a specified value (that is, a value not computed in the usual way by DPCS).

The subsections of this section tell how to perform each of these separate tasks by using PSUB (if the job is new) or PALTER (if the job has already been submitted). To be authorized to use PSUB and PALTER in these special ways, you must be either:

- a DPCS manager who is also a DPCS "expeditor," or
- a coordinator of a bank that is a parent of the bank from which the job is drawing its resources, and who is also a DPCS "expeditor," or
- a user who owns the job and who has been given permission for a specfied number of days by someone in the previous two authorized groups. Special LRMMGR options (explained in the last subsection (page 52) below) grant these permissions to users.

Until May, 2003, the special users entitled to expedite or exempt jobs were reported by DPCS not as "expeditors" but as having "e" access or "e" permission. Now, using the LRMMGR command

```
show user uname
```

will report "User is a DPCS expeditor" (if they are), while using

```
show expeditor uname
```

will either confirm this status by returning *uname* or instead state "User *uname* is not an expeditor."

Expediting Jobs

HOW TO EXPEDITE.

Expediting a job means giving it such a strong scheduling preference that it starts as soon as possible, even *preempting* other jobs if necessary (see below). Authorized users (page 52) can expedite a batch job by following these steps:

(1) Submit the job as usual by running PSUB, but include the special -expedite option on the execute line as well:

```
psub usualopts -expedite jobname
```

If you are not authorized to expedite jobs, your job will still be accepted by DPCS but the expedite request will be ignored. If you have *already* submitted a DPCS job and then decide you want it expedited, use PALTER as shown below. Currently, DPCS imposes no limit on the number of simultaneous expedited jobs.

(2) Discover the DPCS job ID by running PSTAT (use the -A option to see all jobs if you are not the job's owner).

(3) Use the job ID (*jobid*) in this PALTER execute line to expedite the already-submitted job:

```
palter -n jobid -expedite
```

(4) Similarly, to cancel expedition of a previously expedited job, use this PALTER execute line:

```
palter -n jobid -noexpedite
```

Previously, the PEXP utility expedited jobs. PEXP has been rendered obsolete by the foregoing features of PSUB and PALTER, but for historical consistency you can still use it as in the past. And PEXP users who want to cancel expedition of a previously expedited job can now type

```
pexp jobid -noexpedite
```

On IBM SP computers (only), DPCS can now immediately start an expedited job because of its enhanced ability to preempt running jobs by using refined memory-management features. However, PSUB's -A option (which specifies an earliest start time) dominates the -expedite option, so no job ever starts before its -A time, even if you expedite it.

PREEMPTION CONSEQUENCES.

Expediting one job often affects other running jobs on the machine where DPCS/LCRM starts the newcomer. The DPCS/LCRM goal is to maximize node use. DPCS/LCRM starts the expedited job on free nodes if enough are available. If not, DPCS/LCRM terminates standby jobs until enough nodes are released to start the expedited job. If still more nodes are needed (and if system administrators have enabled preemption), DPCS/LCRM then *preempts* (some) normal jobs to temporarily make their nodes available as well.

When DPCS/LCRM preempts a normal job, then that job:

- halts execution but remains memory resident,
- temporarily releases its nodes for use by the expedited job,
- charges no time (elapsed or CPU) during its preemption pause,

- shows the job status PREEMPTD in PSTAT reports, and
- resumes automatically when the expedited job that borrowed its nodes ends.

To enable such normal-job preemption on an IBM SP (only), the system administrator must (1) set the scheduling mode of LoadLeveler to API, (2) restart the PSPD daemon on the machine's DPCS/LCRM gateway node, and (3) use the LRMMGR utility to specify a suitably large value for "maximum node divergence," the maximum number of nodes that are allowed to go idle as a side effect of scheduling an expedited job over preempted normal jobs. The LRMMGR command to specify maximum node divergence (allowed idle nodes) is

```
set global maxnodediverge n
```

where *n* is a positive integer.

Exempting Jobs

Exempting a job means allowing it to run even if it exceeds administratively imposed constraints (such as on number of CPUs needed or maximun job size) that prevent other jobs from running. These general, systematic constraints are often called DPCS "statuses" to distinguish them from the user-imposed constraints that you specify with PSUB's -c option (and because PSTAT reports the ones that currently block a job from running as the job's "status"). Authorized users (page 52) can exempt a batch job by following these steps:

(1) Submit the job as usual by running PSUB, but include the special -exempt option on the execute line as well:

psub *usualopts* -exempt ['*statuslist*'] *jobname*

where *statuslist* is an optional, single-quoted, comma-delimited list of DPCS statuses (administratively imposed constraints) from which you wish to exempt this job (for example, 'CPUS>MAX,TOOLONG'). The only currently exemptable DPCS statuses are (note the uppercase):

```
CPUS>MAX
CPU&TIME
JRESLIM
NRESLIM
NTRESLIM
QTOTLIM
QTOTLIMU
TOOLONG
WMEML
```

You can omit the single quotes around *statuslist* if the list contains no special characters that need protection from the shell. If you omit *statuslist* entirely, the job is exempt from EVERY status from which you have permission to exempt jobs. See the status list (page 20) section above for an alphabetical dictionary that explains every DPCS status, including the exemptable ones.

If you are not authorized to exempt jobs, your job will still be accepted by DPCS but the exempt request will be ignored. If you have *already* submitted a DPCS job and then decide you want it exempted, use PALTER as shown below.

(2) Discover the DPCS job ID by running PSTAT (use the -A option to see all jobs if you are not the job's owner).

(3) Use the job ID (*jobid*) in this PALTER execute line to exempt the job:

palter -n *jobid* -exempt ['*statuslist*']

where *statuslist* meets the same conditions as in step (1) above.

(4) Similarly, to remove exemption from a previously exempted job, use this PALTER execute line:

palter -n *jobid* -noexempt ['*statuslist*']

where *statuslist* meets the same conditions as in (1) above. If you omit *statuslist* entirely here (use -noexempt without arguments), then the job loses ALL of its previous exemptions (and is subject to all the usual administrative constraints).

Forcing Job Priorities

Forcing a job's priority means assigning a specific value to its execution priority rather than letting the usual DPCS algorithms calculate that priority and change it periodically (forced priorities remain constant for the life of the job). Authorized users (page 52) can force the priority of a batch job by following these steps:

(1) Submit the job as usual by running PSUB, but include the special -p option on the execute line as well:

```
psub usualopts -p priority jobname
```

where *priority* is a value between 0.0 and 1.0 inclusive. Setting the priority to 0.0 will prevent DPCS from scheduling the job. This has the same effect at running PHOLD, except that a 0.0-priority job's aging time continues to advance.
If you are not authorized to force priorities jobs, your job will still be accepted by DPCS but the priority request will be ignored. If you have *already* submitted a DPCS job and then decide you want its priority forced, use PALTER as shown below.

(2) Discover the DPCS job ID by running PSTAT (use the -A option to see all jobs if you are not the job's owner).

(3) Use the job ID (*jobid*) in this PALTER execute line to force the job's priority:

```
palter -n jobid -p priority
```

where *priority* is a value between 0.0 and 1.0 inclusive. Setting the priority to 0.0 will prevent DPCS from scheduling the job. This has the same effect at running PHOLD, except that a 0.0-priority job's aging time continues to advance.

(4) Similarly, to let DPCS once again compute the priority of a previously forced job, use this PALTER execute line:

```
palter -n jobid -p float
```

Note that PALTER's -p option formerly set a job's "intrabank scheduling priority." This feature was never used, and now is completely replaced by the current priority-forcing role for -p. The same applies for PSUB's former -p option.

Granting Special-Job Permissions

To be authorized to use PALTER to expedite jobs, exempt jobs, or force job priorities, you must be either

- a DPCS manager who is also a DPCS "expeditor," or
- a coordinator of a bank that is a parent of the bank from which the job is drawing its resources, and who is also a DPCS "expeditor," or
- a user who owns the job and who has been given permission for a specfied number of days by someone in the previous two authorized groups.

This section tells how to run the LRMMGR utility (formerly called PCSMGR) to grant special job-control permissions to otherwise ordinary users.

First, execute LRMMGR (no options). Then, respond to the lrmmgr> prompt by typing an input line of the form:

```
update user uname bank bname grantperm
```

where

uname is the login name of the user to whom you are granting special job-control permissions.

bname is the name of the bank with which the specified user will exercise their special permissions.

grantperm is one or more of the job-control permissions that you can grant, specified singly or in a blank-delimited list (if you want to grant several permissions on one input line). The choices for *grantperm* are (one or more of the following):

expcount *days* grants the specified user (for the specified bank) the permission to expedite their own jobs with PALTER for the specified number of *days*, where *days* may be--
1 to 14 (an inclusive time range), or
0 (removes previous expedite permission), or
unlimited (a literal string, no time limit).

exemptcount *days* exemptstats *statuslist*

grants the specified user (for the specified bank) the permission to exempt their own jobs with PALTER for the specified number of *days*, where *days* may be--
1 to 14 (an inclusive time range), or
0 (removes previous exempt permission), or
unlimited (a literal string, no time limit).
Here *statuslist* is a single-quoted, comma-delimited list of DPCS "statuses" (administratively imposed limits) for which the user can exempt jobs, as explained in the subsection above (page 50) on how to exempt jobs by running PALTER.

fixpriocount *days* grants the specified user (for the specified bank) the permission to force the priority of their own jobs with PALTER for the specified number of *days*, where *days* may be--
1 to 14 (an inclusive time range), or
0 (removes previous force-priority permission), or
unlimited (a literal string, no time limit).

For example, to use LRMMGR to grant to user jones3 for bank xyz the permission to exempt his jobs from the QTOTLIMU restriction for the next 2 days and the permission to force job priorities forever, but to simultaneously withdraw his previous permission to expedite his jobs, you would use this input line in response to the lrmmgr> prompt:

```
update user jones3 bank xyz exemptcount 2 exemptstats QTOTLIMU
       fixpriocount unlimited expcount 0
```

You can reveal the currently granted permissions for any user and bank combination by using the LRMMGR show command. For example,

```
show user uname
```

reports "User is a DPCS expeditor" (if they are).

Fair Share Scheduling Algorithms

This section explains the concepts (such as shares, normalization, usage decay, and priority), the formulas, and the parameter settings used to implement fair-share job scheduling at LC. Fair-share scheduling replaced traditional time-allocation scheduling on LC open production machines in March, 1998, and then it moved to the SCF production machines as well by June, 1998.

Definitions

Fair-share scheduling is one variety of political scheduling, that is, scheduling aimed at dividing compute resources among users or groups of users (as opposed to dividing jobs among available machines (load balancing) or grouping related tasks to run together (gang scheduling)). Fair-share scheduling as implemented at LC under DPCS involves two key concepts that were unimportant for traditional time-allocation political scheduling: shares and active users.

Shares

Shares are assigned to each user to represent in a unitless way that user's relative entitlement to system resources (primarily CPU time, but eventually other resources such as memory too). A high number of shares relative to other users represents a higher entitlement to compute, and hence a broader range of circumstances when that user gets a high(er) scheduling priority. Conversely, users or banks (groups of users) with similar numbers of shares (similar "share allocations") get to use similar amounts of compute resources, regardless of the number of processes they may have executing. Traditional schedulers tend to allow users with more processes to get a larger percentage of system resources than their priority alone would allow.

Your shares influence the calculation of the scheduling priority for your jobs (see below (page 60)), but they are not a measure of any specific resource (they are not equivalent to some number of CPU seconds, for example). As a result, you never "use up your share" or "run out of time," as is possible under time-based allocations. Your usage influences your job priority too, but it does not deplete your bank account.

Those who manage banks (primarily divisional computer coordinators) assign shares or alter share assignments by running LRMMGR. For example, here LRMMGR's UPDATE option assigns 15 shares to user *aaa* in bank *bbb*:

```
update user aaa bank bbb share 15
```

Any user can run the PSHARE utility to report their currently assigned shares (and the priority those shares help generate), as shown below in the priority section (page 60).

Also, LC shares are hierarchical, in the sense that banks have shares of their parent banks just as users have shares of their direct banks. Compute entitlements are assigned and enforced in layers, just as time allocations were in traditional scheduling. The normalization (page 56) section below gives a worked-out example of this share hierarchy.

Active Users

LC fair-share scheduling "normalizes" both shares and usage among all and only "active users" (in the same bank) in order to calculate priorities. So the definition of an active user is crucial to the numerical result. An active user is one who:

- is currently logged in (even if NOT executing any processes), or
- has at least one batch job running now, or
- has at least one batch job ELIGIBLE to run (where "eligible" is a technical DPCS job status).

An active bank is a bank with at least one active user (who may really be a direct user of some subbank).

Shares and their Normalization

ROLE:

To allow their comparison when computing a user's priority, both the user's raw shares and their raw usage are "normalized" to yield a number between 0 and 1. In a significant departure from traditional scheduling, LC counts only currently active users (as defined in the previous section) when normalizing shares (and usage). This means that merely by logging in or out, users can affect the normalized value of other users's shares. DPCS recalculates normalized values once each minute (sometimes called the "heartbeat" rate).

At LC your normalized share value applies globally, over an entire DPCS partition (e.g., over all machines in the GPS cluster). Machines of different types (e.g., BLUE, LX, GPS) are in different partitions, each with its own normalized share values.

FORMULA:

The LC normalization formula is

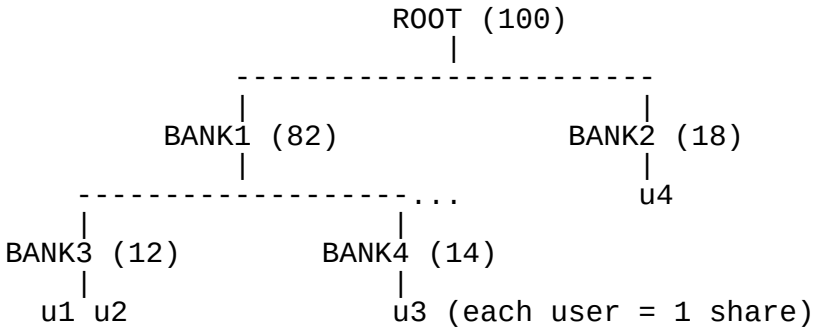
$$nor.val = \frac{A(raw.val)}{\text{SUM } A(raw.val(i))} * (nor.val)_{parent}$$

where

nor.val	is the user's normalized value for shares (or usage).
raw.val	is the user's raw value for shares (or usage).
A(raw.val)	is a step function that filters out nonactive users by returning: raw.val — if the user is active, and 0 — if the user is currently not active (so your normalized shares are 0 whenever you are not active).
raw.val(i)	is the raw value for the ith user in the same bank in the same DPCS partition.
parent nor.val	is the normalized value for the parent (bank) of this user or bank. This formula is always applied recursively up the tree of banks until the root bank (whose normalized value is 1) is reached.

EXAMPLE:

This simple example demonstrates how such recursive normalization of shares among only active users works in practice. Suppose raw shares are allocated among banks and users as shown in this tree:



Then if all and only the four users (u1 through u4) are active, their normalized shares will be

```
u1: (1/2) * (12/26) * (82/100) = 0.19
u2: (1/2) * (12/26) * (82/100) = 0.19
u3: (1/1) * (14/26) * (82/100) = 0.44
u4: (1/1) * (18/100)           = 0.18
```

If user u1 logs out (becomes inactive), then user u2's normalized share doubles to 0.38 (assuming no other changes). Normalized values are recalculated once each minute.

To see the actual hierarchy of banks relevant (for normalization) on any LC production machine (open or secure), log on to that machine and type

pshare -T root

(this yields a very long report, which now shows the full names of every bank in the hierarchy).

Usage and Its Decay

For purposes of fair-share scheduling, the usage of each user is "aggregated" across all compute resources and also across the user's historical profile.

Usage (in "aggregate resource units" or AGUs) is the weighted measure of compute resources consumed, including:

- CPU time,
- Memory integral (currently weight set to 0), and
- Connect time (currently weight set to 0).

So for now, only CPU time actually contributes to fair-share usage. Taking this aggregate approach helps prevent users with many active processes from consuming CPU resources at a higher rate than those users with only a few active processes.

One goal of fair-share scheduling is to favor users who have relatively more shares with relatively more compute resources. A second goal is to fairly distribute resources among those users who have equal shares. This second goal is achieved by tracking usage, not just instantaneously but over time. Tracking usage history allows the scheduler to allocate relatively more resources to a user who has done less computational work in the recent past than to one who has done more work. This also promotes the system-management goal of spreading work rather than crowding it whenever possible.

The length of time during which your past work affects your priority (and hence the scheduling of your future jobs) is specified by a decay factor. At LC, that decay factor is expressed as a half-life of usage history, an interval over which your usage would drop to half its value (if all other things were constant). The specific formula for decaying your usage at LC is

$$\text{current decayed usage} = \frac{U}{2^{(dt/dR)}} + \frac{W}{2^{(dt/2dR)}}$$

where

U	is past (previously decayed) usage.
W	is new usage ("work") done during the interval dt.
dt	is the time interval between usage samples. At LC, dt may vary depending on machine load and machine type. Current value of dt: 540 seconds
dR	is the half-life decay period for usage. Experiments with this formula show that larger dR values reduce errors and yield more stable priorities than do smaller dR values. Current value of dR: 1 week

To calculate priorities, (decayed) usage and shares must be compared, and so both are normalized using the approach explained in the previous section (page 56). The usage that matters to the priority calculation

is thus both a decayed historical aggregate and normalized over current active users. Consequently, it has little direct connection to the raw reports of CPU minutes used that the utility PCSUSAGE delivers. Also, PCSUSAGE always reports time used by whole day, and days have no role in calculating priorities for fair-share scheduling.

Priority Calculation

FORMULA:

Different fair-share systems calculate job priorities in different ways. LC uses a priority formula that

- maps all priorities into the range from 0 to 1, inclusive,
- regards priority 0.5 as "neutral," indicating resource consumption neither ahead of nor behind what a user's share entitles,
- calculates one priority per user throughout each DPCS "resource partition" (e.g., one for each Compaq cluster and each IBM SP),
- relies only on differences, not on absolute values (of normalized shares and usage) to compute priorities (see comments below).

The current LC fair-share priority formula is

$$P = [(S * W) + \frac{((S - U) + 1)R}{2}](1 - E) + [1 - 2^{(-S/U)}]E$$

where

P	is the "political" priority (a decimal number between 0 and 1 inclusive). See the next subsection for how this priority affects job scheduling.
S	is normalized shares (also between 0 and 1, as explained in the normalization (page 56) section above). Sometimes also called "share priority."
U	is normalized half-life decayed usage (also between 0 and 1, as explained in the usage (page 58) section above). Sometimes also called "usage priority."
R	is a configurable weighting factor for usage (sometimes called uweight). Current value of R: 1
W	is a configurable weighting factor for shares (sometimes called sweight). Current value of W: 0
E	is a configurable weighting factor for the exponential (right-hand) term in this equation (sometimes called eweight). Current value of E: 0

Thus currently the crucial part of the priority formula is the central fraction (S-U+1)/2, whose value is more important than either normalized shares or normalized usage in isolation. (See CONSEQUENCES below for the role of the exponential term, now set to 0 because of the value of E.)

EXAMPLE:

We can extend the share normalization example in the normalization (page 56) section to become an example of priority calculation using this formula (simplified to (S-U+1)/2 given the current weights) if we first stipulate some normalized decayed usage values for each user:

```
        Norm.    Norm.    Resulting
User    share    usage    priority

 u1     0.19     0.6      0.295
 u2     0.19     0.2      0.495
 u3     0.44     0.1      0.670
 u4     0.18     0.1      0.540
```

PSHARE REPORTS:

Using the -p option of the PSHARE utility reports actual current priorities, along with the normalized share and usage values that gave rise to them. (Remember that the normalized values for NONactive users are always defined as zero.) This PSHARE execute line

pshare -t *yourbank* -0 -p

is especially helpful because it reports priorities (-p) for all (-t) but only (-0) the currently active users in your own bank. This is the most interesting and most relevant comparison set for your own priority when planning jobs. PSHARE calculations (normalizations) are refreshed once each minute.

CONSEQUENCES:

One noteworthy consequence of this approach to calculating "political" priority is that changes in the set of active users change the normalization, and hence can change, sometimes drastically, the priority values assigned to other users. In the example above, recall that users u1 and u2 are in the same bank. If user u1 (who had much accumulated usage) logs out, then the priority for user u2 will be recalculated to become 0.29, a significant drop. This dependence of (normalization and hence) priority on the set of users currently seeking resources is an major change from past practice, where time-allocation priorities were independent of the active user set.

A second noteworthy consequence of this priority formula is the effect of the configurable weights R, W, and E. With W and E set to 0 and R set to 1 (the curent defaults), the formula ignores the absolute value of normalized shares and usage, and relies only on the difference in magnitude between them (S - U). If everyone in a bank has about the same number of shares, the difference results are a plausible interpretation of "fairly" sharing resources. If the variation among shares is great (e.g., 10-fold), however, then it becomes possible for small shareholders to always have (relatively) low priorities, even though large shareholders with exactly the same difference value (S - U) have already consumed large amounts of compute resources equalling a large percentage of their entitlement. In these cases "fairness" may have several dimensions that a purely difference formula overlooks.

This artificial emphasis on difference explains the presence of the right-hand, exponential term in the formula, which was added in December, 2000. The term 1-2**(-S/U) takes account of the ratio of normalized shares to normalized usage, not just their absolute difference. If exponential weight E were set to 1 instead of 0 (the default), then the (S-U) difference would become unimportant. Small, heavily serviced banks would more often have low priorties compared to larger, less seviced banks (in fact, the danger here is that small bank priorities would stay so low that their jobs would never be scheduled).

SETTING WEIGHTS:

Beginning in December, 2000, the three weight factors in the priority formula above, namely

```
R  (uweight)  usage weight
```

```
W  (sweight)  share weight
E  (eweight)  exponential weight
```

can be set (by DPCS managers or authorized bank coordinators) to different values for different DPCS "resource partitions" (such as for each Compaq cluster and for each IBM SP machine). Authorized LRMMGR users can set each weight independently by replying to the lrmmgr> prompt with an input line of the form

```
update partition pname uweight weightval
                       sweight
                       eweight
```

where *pname* is the target partition's name and *weightval* is a decimal number between 0 and 1 inclusive. The previously available

```
update global uweight
              sweight
```

LRMMGR commands are now obsolete and will yield only error messages if tried.

Role of Priority in Job Scheduling

In theory the fair-share priority calculated using the formula in the previous section plays a dual role in managing jobs on LC production machines:

- Scheduling priority.
  This helps determine which queued batch jobs should run next.
- Run priority.
  This helps determine the rate of delivery of resources to login sessions and batch jobs already underway. Since the current mechanism for controlling delivery rate is nice value, this role has little significance now. Under a gang scheduler that controlled time sharing (as well as space sharing), this could become more important in the future.

Just as in the past, the algorithm for which job is scheduled next is complex. Priority is a key factor (reflecting as it does the influence of both shares and usage). But many other factors (such as a machine's maximum number of simultaneous jobs and a user's maximum number of simultaneous jobs) also affect the outcome. The underlying DPCS job-scheduling algorithm (page 37), described in another section, remains as it was before the introduction of the fair-share approach. But now your fair-share priority serves as your "political priority" pp[j] when the algorithm is invoked.

Graceful Priority-Service Transition

Warning Alternatives

This section explains extended features that allow DPCS to gracefully terminate executing jobs (and passively or actively warn those jobs) on a running system slated

- for dedicated ("priority-service") usage, or
- to move from one kind of usage to another (e.g., from batch intensive to interactive intensive).

GENERAL APPROACH:
This approach involves a change to a DPCS/LCRM administrative command and three new library calls:

(1) The change to the existing "update host" command for the LRMMGR utility adds an optional effective time at which a priority service level is to take effect. In the absence of a specified effective time, this command reverts to its existing behavior, which is to place the host into the specified priority service immediately.

(2) One of the new library calls, pcsgetresource (also called lrmgetresource), permits a program to determine the time at which a priority service will become effective, if the job would be terminated or checkpointed by the advent of the priority service. (Note: the original proposal concerning priority service indicated that the pcsgettime call would be modified for this purpose. But the addition of a number of parameters concerning memory usage has motivated the introduction of the completely new call, pcsgetresource. This avoids having an impact on programs that currently use the pcsgettime call, and it also expresses the functionality of the call more clearly.)

(3) The two new library functions, pcssig_register (also called lrmsig_register) and pcswarn (also called lrmwarn), permit a program to register with DPCS to be sent a signal when DPCS is instructed to set a priority service that would cause the job to be checkpointed or terminated. Code developers should use pcssig_register if they want to trap the signal. They should use pcswarn if they wish to poll a variable that indicates whether the priority service that would affect the job has been declared.

CURRENT STATUS:
The warning strategy outlined above and described in detail below was implemented first on the open LC machines (Compaq clusters and Blue) in March, 1998. It then migrated to the SCF production machines after successful testing in the open environment (June, 1998).

Library Calls

This section describes the three library calls (functions) that implement graceful priority-service transitions.

As a programmer, you should use either pcssig_register/IPCSSIG_REGISTER or pcswarn/IPCSWARN, but not both (you can also invoke these with the names lrmsig_register or lrmwarn, as illustrated below).

If pcswarn/IPCSWARN is used, you do not need to supply a signal handler, but rather should poll on the *warn/WARN flag and *stoptime/STOPTIME variable to determine the code's proper action.

If pcssig_register/IPCSSIG_REGISTER is used, you should register the appropriate signal handler to trap the signal and should call pcsgetresource/IPCSGETRESOURCE (also called lrmgetresource) directly to determine the reason the signal was sent.

PCSGETRESOURCE (LRMGETRESOURCE)

NAME:

```
pcsgetresource   (or lrmgetresource, called from C)

IPCSGETRESOURCE  (called from FORTRAN)
```

SYNOPSIS:

```
#include <libpcs.h>
#include <pcserrno.h>

int pcsgetresource(time_t *total, time_t *used, time_t
*maxtime, time_t *avail, time_t *stoptime, long *arus, long
*maxarus, double *memint, double *maxmemint, int *pcsstatus);

INTEGER IERR, TOTAL, USED, MAXTIME, AVAIL, STOPTIME
IERR = IPCSGETRESOURCE(TOTAL, USED, MAXTIME, AVAIL,
STOPTIME,ARUS,MAXARUS,MEMINT,MAXMEMINT)
```

pcsgetresource() and IPCSGETRESOURCE return several resource-related values in buffers provided by the caller. All time values are in seconds. If your program is designed to terminate gracefully rather than being shutdown by the system, the following values should be examined: stoptime (STOPTIME), avail(AVAIL), and the difference between arus(ARUS) and maxarus(MAXARUS). The program should also take into consideration the time and resources required to archive results, if appropriate.

*total (TOTAL)

contains the total CPU seconds used by the session.

*used (USED) contains the CPU seconds used since the session last began execution. (This will differ from *total only in batch jobs that have been checkpointed).

*maxtime (MAXTIME)

contains the maximum amount of CPU seconds per task permitted to the session. If maxtime is unlimited, -1 will be returned for this parameter. (Except on the IBM SP machines, there is only one task per job.)

*avail (AVAIL)

contains the amount of remaining CPU seconds available to the session. The remaining CPU time available to a session (*avail or AVAIL) is the instantaneous value only. Some or all of the time reported as available may be used by other users drawing from the same bank subtree.

*stoptime (STOPTIME)

contains 0 if no priority service has been declared or if the job is protected by a declared priority service. Otherwise it contains the local time (seconds since midnight, January 1, 1970 UTS) at which the job will be checkpointed or terminated. The time function reports the current time for comparison.

*arus (ARUS) is the quantity of "Aggregate Resource Units" used by the session (Note: ARU is an as-yet unspecified quantity that will be used to unify memory and cpu charging when memory charging is implemented.)

*maxarus (MAXARUS)

is the total amount of ARUs available to the session. If maxarus is unlimited, -1 will be returned for this parameter.

*memint (MEMINT)

is the memory integral (kilobyte seconds) used by the session.

*maxmemint (MAXMEMINT)

is the total memory integral (kilobyte seconds) available to the session. If maxmemint is unlimited, -1 will be returned for this parameter.

ERROR CONDITIONS:

If pcsgetresource fails it returns -1 and *pcsstatus (page 69) contains a value that indicates the error condition. Otherwise, it returns 0 and *pcsstatus contains 0. If IPCSGETRESOURCE fails, IERR is set to a nonzero value that indicates the error condition. Otherwise, IERR is set to 0.

PCSSIG_REGISTER (LRMSIG_REGISTER)

NAME:

```
  pcssig_register   (or lrmsig_register, called from C)

  IPCSSIG_REGISTER  (called from FORTRAN)
```

SYNOPSIS:

```
#include <libpcs.h>
#include <pcserrno.h>

int pcssig_register(int signal, time_t mintime, int
*pcs_status);

INTEGER IERR, SIGNAL, MINTIME
IERR = IPCSSIG_REGISTER(SIGNAL, MINTIME)
```

pcssig_register() and IPCSSIG_REGISTER register the calling process as being the recipient of the given signal (SIGNAL) on detection of a "nearing time limit" or "shutdown pending" event for the session (or job) of which the calling process is a member.

A "nearing time limit" event occurs when the remaining CPU time available to a session due to a DPCS imposed limit becomes less than the specified mintime (MINTIME), which is expressed in seconds. A "shutdown pending" event occurs when an administrator specifies to DPCS that the host on which a session is executing is to be placed into priority service at the present or a future time and the session or job will be checkpointed or terminated at the effective priority service time as a result of not being in the priority protected set of sessions. (A session is priority protected if it is drawing its allocated resources from a priority protected bank. A bank is priority protected if it is a sub-bank of the priority bank specified by the administrator when the priority service level was declared.) The program should also take into consideration the time and resources required to archive results, if appropriate.

If either condition is true at the time of the call to pcssig_register() or IPCSSIG_REGISTER, the signal will be sent immediately. The signal is also sent to a registered process when either condition becomes true. When a signal is sent, the registration is deleted. If a process wishes to receive additional signals, it must call pcssig_register() or IPCSSIG_REGISTER again.

If more than one process in a session calls either pcssig_register() or IPCSSIG_REGISTER, then only the last process that makes either call will receive a signal from DPCS. No process will be notified that it will not receive the signal if it is preempted by another process. If, from among all the processes of a session, the process that has last called pcssig_register() or IPCSSIG_REGISTER terminates before the signal is sent, then no signal is sent to any process of the session unless another process of the session subsequently calls either routine.

The specified signal can not be SIGKILL or SIGSTOP (or any other signal that cannot be caught).

It is the responsibility of the caller to register a signal handler with the operating system to trap the signal when it is sent.

ERROR CONDITIONS:

If pcssig_register fails it returns -1 and *pcsstatus (page 69) contains a value that indicates the error condition. Otherwise, it returns 0 and *pcsstatus contains 0. If IPCSSIG_REGISTER fails, IERR is set to a nonzero value that indicates the error condition. Otherwise, IERR is set to 0.

PCSWARN (LRMWARN)

NAME:

```
pcswarn   (or lrmwarn, called from C)

IPCSWARN  (called from FORTRAN)
```

SYNOPSIS:

```
#include <libpcs.h>
#include <pcserrno.h>

int pcswarn(int signal, time_t mintime, int *warn, time_t
*stoptime, int *pcsstatus);

INTEGER IERR, SIGNAL, MINTIME, WARN, STOPTIME
IERR = IPCSWARN(SIGNAL, MINTIME, WARN, STOPTIME)
```

pcswarn() and IPCSWARN store the value 0 into *warn (WARN) and *stoptime (STOPTIME). The functions then register an internal signal handler with the operating system to trap the specified signal (SIGNAL). Finally, they call pcssig_register() to register the signal with DPCS. When the signal is sent, the internal signal handler calls pcsgetresource to get the value to store into *stoptime (STOPTIME) and then sets *warn (WARN) to 1 if and only if the job's available time is less than or equal to mintime (MINTIME), which is expressed in units of seconds. The program should also take into consideration the time and resources required to archive results, if appropriate.

ERROR CONDITIONS:

If pcswarn fails it returns -1 and *pcsstatus (page 69) contains a value that indicates the error condition. Otherwise, it returns 0 and *pcsstatus contains 0. If IPCSWARN fails, IERR is set to a nonzero value that indicates the error condition. Otherwise, IERR is set to 0.

Error Conditions (*pcsstatus)

Possible errors from the three foregoing library functions appear in the list below. Failures cause *pcsstatus to contain a value that indicates the error condition. The status value (or return value from FORTRAN extensions) is identified in the file pcserrno.h, which is located in /usr/local/include (and the value also appears in parenthesis in this list).

PCS_EINVAL (5001)

Invalid parameter value was found.

PCS_ENOHOST (5050)

The caller is executing on a host that is not being managed by DPCS.

PCS_ENOSID (5018)

DPCS did not find the caller's session.

PCS_ENOTOPEN (5031)

Can't open communication with DPCS daemon, not connected.

PCS_EREADERR (5037)

Error reading from DPCS daemon.

PCS_ERETRY (5011)

Action could not be completed. Retry.

PCS_ESELERR (5036)

Select error on DPCS daemon return socket.

PCS_EUIDRANGE (5063)

The user to be affected has a UID that is not in the range of UIDs being managed by DPCS.

PCS_EWRITEERR (5035)

Error writing to DPCS daemon.

Examples

Poll-for-Warning Examples

1. POLLING IPCSWARN (FORTRAN).

Assume a FORTRAN-coded program with a major cycle that takes no more than 50 CPU minutes to complete each interation and that the program requires 5 CPU minutes to gracefully terminate. To register this process to receive a signal with sufficient time for gracefully termination prior to a DPCS initiated termination or checkpoint, the code developer would add the following lines into the code before entering the major cycle:

```
c       Register with DPCS to give a warning if the
c       available CPU time becomes less
c       than 1 hour or if a priority service that would
c       cause the job to terminate becomes
c       effective.

        IERR = IPCSWARN(SIGNAL, 60*60, WARN, STOPTIME)
```

In this example, the program would then enter its major cycle. At the beginning of the cycle, the code should check the values of WARN and STOPTIME. If both WARN and STOPTIME are 0, then the code can continue its major cycle with relatively strong assurance that it can be completed. If WARN is not 0, the code should enter its graceful termination code, after which it can either terminate or wait to be checkpointed. If WARN is 0, but STOPTIME is not, the programmer should determine the appropriate action from the wall clock time remaining to the job.

2. POLLING PCSWARN (C).

A sample C program using the pcswarn function (called here as lrmwarn) is shown below. This sample program does the same as the one in the next section (page 75), except it is using the pcswarn function instead of the pcssig_register function. This program waits a designated amount of time before terminating (default is 60 CPU seconds).

WARNING: You must link in the DPCS library, -lpcs (or -llrm, /usr/local/lib/libpcs.a), when compiling this sample program.

```
#include <signal.h>

#include <liblrm.h>

#include <lrmerrno.h>

#include <time.h>


/*

 *  cc -o lrmwarnexample lrmwarnexample.c -L/dpcs/lib -llrm -I/dpcs/include

 */


void display_resource_info(void)
```

```
{
 long total = 0;
 long used = 0;
 long maxtime = 0;
 long avail = 0;
 long stoptime = 0;
 long arus = 0;
 long maxarus = 0;
 long memint = 0;
 long maxmemint = 0;
 int lrmstatus = 0;
 int rc = 0;
 char str[64];

 rc = lrmgetresource(&total, &used, &maxtime, &avail, &stoptime, &arus,
       &maxarus, &memint, &maxmemint, &lrmstatus);
 if (rc == 0) {
  printf("\tTotal CPU seconds:     %ld\n", total);
  printf("\tConsecutive CPU secs:  %ld\n", used);
  sprintf(str, "%ld", maxtime);
  printf("\tMax CPU secs limit:    %s\n",
         maxtime == -1 ? "unlimited" : str);
  sprintf(str, "%ld", avail);
  printf("\tRemaining CPU secs:    %s\n",
         avail == -1 ? "unlimited" : str);
  printf("\tStoptime               %s",
         stoptime <= 0 ? "N/A\n" : ctime(&stoptime));
  printf("\tAggregate Resrc Units: %ld\n", arus);
  sprintf(str, "%ld", maxarus);
  printf("\tARU limit:             %s\n",
```

```
        maxarus == -1 ? "unlimited" : str);
  printf("\tMemory integral:       %ld MB-hours\n", memint);
  sprintf(str, "%ld", maxmemint);
  printf("\tMemory integral limit: %s MB-hours\n\n",
        maxmemint == -1 ? "unlimited" : str);
 } else {
  printf("lrmgetresource() Failed!\n");
  printf("    return code = [%d] lrmstatus = [%d]\n",
        rc, lrmstatus);
 }
 return;
}

static void burn_cpu(void)
{
 double l = 456.789;
 int i;

 for (i = 0; i < 100000000; i++) {
  l *= 123.456 * i;
  l /= 123.456 * i;
 }
}

int main(int argc, char *argv[])
{
 long  mintime;
 int  warning = 0;  /* NOTE: this will be set by lrmwarn() */
 long  stoptime = 0; /* NOTE: this will be set by lrmwarn() */
 int  lrmstatus = 0;
 time_t  Now;
```

```
display_resource_info();

if (argc == 2)
 mintime = atoi(argv[1]);
else
 mintime = 60;

if (lrmwarn(SIGINT, mintime, &warning, &stoptime, &lrmstatus)) {
 printf("lrmwarn() failed to register SIGINT with LCRM\n");
 printf("    lrmstatus = [%d]\n", lrmstatus);
 exit(1);
} else {
 printf("Requested a warning when %ld CPU secs remain\n",
        mintime);
}

/* burn some cpu cycles while waiting for the warning */

while (!warning && !stoptime) {
 time(&Now);
 printf("waiting for the warning to be received... %s\n",
        ctime(&Now));
 burn_cpu();
}

printf("warning = [%d]  stoptime = [%ld]\n", warning, stoptime);
if (stoptime)
 printf("Stop time = %s\n", ctime(&stoptime));
else
 printf("Stop time = normally\n\n");

display_resource_info();
```

```
 return(0);
}
```

Signal-Catching Examples

1. BRIEF SIGNAL CHECK (C).

Assume a C-coded program with a major cycle that takes no more than 50 CPU minutes to complete each interation and that the program requires 5 CPU minutes to gracefully terminate. Also, assume that the programmer does not wish to simply poll, but requires immediate notification when the signal is sent. The code developer would place the following lines into the code before entering the major cycle:

```
static void mypcssig_handler(int sig)
{
     time_t    total, used, maxtime, avail, stoptime;
     int  pcsstatus;
     long      arus, maxarus;
     double memint, maxmemint;

     if (!pcsgetresource (&total, &used, &maxtime, &avail,
&stoptime,
                              &arus, &maxarus, &memint,
&maxmemint,
                              &pcsstatus))

     {
          /* process the DPCS error */
     } else
     {
          /* Do what ever is necessary here to handle the
receipt of the signal */
          signal(sig,mypcssig_handler);
          /* might want to do a longjmp here */
     }

     return;
}

...

int main(int argc, char **argv)
{
     ...
     signal(SIGALRM, mypcssig_handler);
     if (!pcssig_register(SIGALRM, mintime, &pcs_status)) {
          /* Handle pcs error */
     }

     ...

     return(0);
}
```

2. ELABORATE SIGNAL CHECK (C).

A more complex sample C program using the lrmsig_register and lrmgetresource ("new" named) functions is shown below. This program waits a designated amount of time before terminating (default is 60 CPU seconds).

WARNING: You must link in the DPCS library, -lpcs (or -llrm, /usr/local/lib/libpcs.a), when compiling this sample program.

```
#include <signal.h>
#include <liblrm.h>
#include <lrmerrno.h>
#include <time.h>

/*
 *  cc -o sigregexample sigregexample.c -L/dpcs/lib -llrm -I/dpcs/include
 */

static int interrupted;

void display_resource_info(void)
{
 long total = 0;
 long used = 0;
 long maxtime = 0;
 long avail = 0;
 long stoptime = 0;
 long arus = 0;
 long maxarus = 0;
 long memint = 0;
 long maxmemint = 0;
 int lrmstatus = 0;
 int rc = 0;
 char str[64];

 rc = lrmgetresource(&total, &used, &maxtime, &avail, &stoptime, &arus,
       &maxarus, &memint, &maxmemint, &lrmstatus);
 if (rc == 0) {
  printf("\tTotal CPU seconds:     %ld\n", total);
  printf("\tConsecutive CPU secs:  %ld\n", used);
  sprintf(str, "%ld", maxtime);
```

```
  printf("\tMax CPU secs limit:    %s\n",
         maxtime == -1 ? "unlimited" : str);
  sprintf(str, "%ld", avail);
  printf("\tRemaining CPU secs:    %s\n",
         avail == -1 ? "unlimited" : str);
  printf("\tStoptime               %s",
         stoptime <= 0 ? "N/A\n" : ctime(&stoptime));
  printf("\tAggregate Resrc Units: %ld\n", arus);
  sprintf(str, "%ld", maxarus);
  printf("\tARU limit:             %s\n",
         maxarus == -1 ? "unlimited" : str);
  printf("\tMemory integral:       %ld MB-hours\n", memint);
  sprintf(str, "%ld", maxmemint);
  printf("\tMemory integral limit: %s MB-hours\n\n",
         maxmemint == -1 ? "unlimited" : str);
 } else {
  printf("lrmgetresource() Failed!\n");
  printf("    return code = [%d] lrmstatus = [%d]\n",
         rc, lrmstatus);
 }
 return;
}

void sigcatch(int sig)
{
 printf("Signal %d received\n", sig);
 interrupted = 1;

 return;
}
```

```
static void burn_cpu(void)
{
 double l = 456.789;
 int i;

 for (i = 0; i < 100000000; i++) {
  l *= 123.456 * i;
  l /= 123.456 * i;
 }
}

int main(int argc, char *argv[])
{
 long  mintime;
 int  lrmstatus = 0;
 time_t  Now;

 display_resource_info();

 if (argc == 2)
  mintime = atoi(argv[1]);
 else
  mintime = 60;

 signal(SIGTSTP, sigcatch);
 if (lrmsig_register(SIGTSTP, mintime, &lrmstatus)) {
  printf("lrmsig_register() failed to register for signal\n");
  printf("    lrmstatus = [%d]\n", lrmstatus);
  exit(1);
 } else {
  printf("Requested a signal when %ld CPU secs remain\n",
         mintime);
```

```
 }

 /* burn some cpu cycles while waiting for the signal */

 interrupted = 0;
 while (!interrupted) {
  time(&Now);
  printf("waiting for the signal to be received... %s\n",
         ctime(&Now));
  burn_cpu();
 }

 display_resource_info();

 return(0);
}
```

Administrative Examples

1. USING LRMMGR.

To place machine X into an urgent priority service for the benefit of bank "eng" at 5:00 p.m. today, an administrator would issue the following command to LRMMGR:

```
update host X psl urgent psbank eng psefftime 17:00
```

To place machine X into an critical priority service immediately:

```
update host X psl critical psbank eng
```

After a machine has been placed into priority service, the service can be removed at a future time. For instance, to remove machine X (which is in priority service) from priority service at 6:00 a.m., an administrator would issue the following command to LRMMGR:

```
update host X psl normal psefftime 06:00
```

Checkpointing

Checkpointing Overview

Checkpointing means saving the state of a running program so that it can continue execution (can restart) later if it is prematurely stopped. There are two primary ways to perform checkpointing, program directed and automatic. In program-directed checkpointing the program saves sufficient state information to continue execution. In automatic checkpointing the operating system or libraries save the program's state. Automatic checkpointing is unable to distinguish between critical state information and temporary storage, which typically results in much more information being recorded than is useful. Automatic checkpointing also has a limited ability to fully restore a program's state. Process IDs, pipes, and data-file state can not always be restored. Programs dependent upon such state information may be unable to utilize automatic checkpointing.

Livermore Computing once offered automatic checkpointing on Cray J90 computers utilizing the UNICOS operating system, without program modification. Automatic checkpointing is now also offered on Compaq (formerly DEC) computers, but only by invoking the Condor libraries. Since operating system support is not offered by the underlying Tru64 UNIX, this mechanism has more restrictions than the former Cray version. The Condor approach also requires the program to be loaded with the appropriate options and libraries. Program-generated checkpoint files are applicable on virtually any computer system, but do require program modification.

Condor Automatic Checkpoint

Condor is a job-scheduling system developed by the University of Wisconsin at Madison that includes a checkpoint mechanism. This checkpoint mechanism is available independently of the job scheduler and has been incorporated into LSF (Load Sharing Facility), GRD (Global Resource Director), Codine, and is planned for our own DPCS (Distributed Production Control System). As one might expect, the checkpoint library utilizes some very unusual constructs:

- A library routine is started prior to the initiation of your "main" routine.
- Signal handlers are established to save the program's state.
- Open, read, write, and close system calls are replaced with Condor versions.

Significant limitiations exist for the type of program that can be built in such a fashion and produce a usable checkpoint image. To build your program in the appropriate fashion, precede the usual execute line(s) for compiling or loading with the string "condor_compile". For example

```
cc -o test test.c
```

would be changed to

```
condor_compile cc -o test test.c
```

When initiating your program, you would add the option

```
-_condor_ckpt filename
```

to identify the location of a file into which the checkpoint image should be written. When restarting your program, add the option

```
-_condor_restart filename
```

to identify the location from which the checkpoint image should be read. For example

```
NORMAL EXECUTION:      my_proc -xyz
CHECKPOINT EXECUTION:  my_proc -_condor_ckpt my_checkpoint -xyz
CHECKPOINT RESTART:    my_proc -_condor_restart   my_checkpoint
```

These Condor options are not passed to your program.

Program checkpoint images will be written upon receipt of a SIGUSR2 or SIGTSTP signal. The SIGUSR2 signal will generate a checkpoint image and continue the program's execution. The SIGTSTP signal will generate a checkpoint image and terminate the program. The program PERIOD_CKPT will automatically generate periodic SIGUSR2 signals to maintain recent checkpoint images. For more information consult the checkpointing MAN page on any of the Compaq clusters of computers (OCF or SCF).

Program-Generated Checkpoint

If you utilize program-generated checkpoints, Livermore Computing advises that they be generated upon receipt of a signal or at periodic intervals. Generating a checkpoint upon receipt of a signal permits the system scheduler to gracefully terminate a job prior to scheduled system down times or if resources need to be released for other purposes. For compatability with Condor, the prefered signals and their meanings are:
SIGUSR2: Generate a checkpoint image and continue job execution
SIGTSTP: Generate a checkpoint file and terminate the job with an exit code 159.

A sample C program to perform checkpointing is shown below.

```
#include <errno.h>
#include <signal.h>
#include <stdio.h>

#define PROB_SIZE 1000
static int array[PROB_SIZE];

void sig_check(int signal_value);
void checkpt_restore(char *checkpoint_filename);

main(int argc, char *argv[]) {
    int i, j;
```

```
    /* Configure for checkpoint generation on signal */
    signal(SIGTSTP, sig_check);
    signal(SIGUSR2, sig_check);

    if ((argc >2) && (strcmp(argv[1], "-restart")== 0)) {
       /* Restore state */
       checkpt_restore(argv[2]);
    } else {
       /* Initialization */
       for (i=0; i<PROB_SIZE; i++) {
          array[i] = i;
       }
    }

    /* Do our work */
    j = 0;
    for (i=0; i<PROB_SIZE; i++) {
       j += array[i];
    }
    printf("Array sum = %d\n", j);

    exit(0);
}  /* main */

/* Generate a checkpoint image */
void sig_check(int signal_value) {
    char checkpoint_filename[30];
    FILE *checkpoint_file;
    static int iteration = 0;
    int err;

    sprintf(checkpoint_filename, "checkpoint.%d.%d", getpid(), iteration++);
```

```
    checkpoint_file = fopen(checkpoint_filename, "w");
    if (checkpoint_file == NULL) {
       fprintf(stderr, "Error %d opening file %s\n", errno, checkpoint_filename);
       return;
    }

    err = fwrite(array, sizeof(int), PROB_SIZE, checkpoint_file);
    if (err != PROB_SIZE) {
       fprintf(stderr, "Error %d writing file %s\n", errno, checkpoint_filename);
    }

    fclose(checkpoint_file);
    if (signal_value == SIGTSTP) exit(159);
    signal(SIGUSR2, sig_check);  /* re-establish signal handler */
}  /* sig_check */

/* Restore a checkpoint image */
void checkpt_restore(char *checkpoint_filename) {
    FILE *checkpoint_file;
    static int interation = 0;
    int err;

    checkpoint_file = fopen(checkpoint_filename, "r");
    if (checkpoint_file == NULL) {
       fprintf(stderr, "Error %d opening file %s\n", errno, checkpoint_filename);
       exit(1);
    }

    err = fread(array, sizeof(int), PROB_SIZE, checkpoint_file);
    if (err != PROB_SIZE) {
       fprintf(stderr, "Error %d reading file %s\n", errno, checkpoint_filename);
       exit(1);
```

```
    }

    fclose(checkpoint_file);
}  /* checkpt_restore */
```

A DPCS Resubmitting Script

To take full advantage of checkpointing, you may want DPCS to automatically restart a program upon system failure. One way to do this is for a DPCS script to submit a restart job that will not begin execution until the original program terminates. This restart job can submit another restart job and so forth to insure eventual completion even should multiple system failures occur as shown in the example below.

```
#! /bin/csh
#
#PSUB -nr  # IMPORTANT, this job should not be re-run !
#PSUB -mb  # mail at the beginning of run.
#PSUB -tM 60:00   # time limit of 60 hours.
#PSUB -r testcode # request name.
#
# This script is resubmitted to DPCS as a new job to be dependent
# upon the completion of this job. The job id is saved so that if this
# job is terminated by anything other than a checkpoint, the dependent
# can be deleted.
#
# The first action should be to set this job up to automatically
# run the job again if the job is checkpointed. Be SURE to use the
# -nr option. This will keep the job from being re-run  if the machine
# should re-boot or the batch system is re-initialized.
#
#  (This example assumes execution in the home directory.)
#

set jobid = `psub -nr -d $PCS_REQID this_scriptname | cut -d' ' -f2`

#
#  Error  recovery if job did not submit properly.
#

If ($status != 0) then
```

```
mailx joe_user -s job submission failure << EOF
*************************************************
re-submission of DPCS job failed from job $PCS_REQID.
*************************************************
EOF
endif

#########################################################################
#
# For testing purposes, the condor method of checkpointing a job was used.
# The executable was made using the following line.
#
#  condor_compile cc -o mytest mytest.c
#
#########################################################################
#
#  Next a checkpoint file name is selected.
#

set ckpt_filename = "mytest.ckpt"

#
# If the checkpoint file exists, this is a restart, otherwise an
# initialization.

if (-e $ckpt_filename)
 ./mytest -_condor_restart $ckpt_filename
else
 ./mytest -_condor_ckpt $ckpt_filename
endif

#
```

```
#  If this point is reached, save the status.
#
set save_exit_value = $status
#############################################################################
#
# NOTE: IF writing your own checkpoint code, make sure that your code
# terminates with  an exit value that truly represents its status (try
# to use an uncommon exit status value to avoid exit status value conflict
# with existing codes).
#
# If you are using the condor checkpointing facility, it has an exit
# status value of 159 which is returned after the code is sent a SIGTSTP
# and the code has checkpointed successfully.
#
#############################################################################
#
#  Set the value of what a successfull checkpoint exit status should be.
#
set checkpoint_value = 159
#############################################################################
#
# Clean up of perpetual job.
#
# If this point is reached, the job has NOT been removed by PRM
# nor deleted by the system. The job has reached completion, either by
#
#     completing successfully, or
#  terminating prematurely  due to error, or
```

```
#  terminating  due to a checkpoint.
#
###############################################################################
#
#  If the job was not checkpointed, it should be deleted.
#

if ($save_exit_value != $checkpoint_value) then

 prm -n $jobid -f

endif
```

Disclaimer

Keyword Index

To see an alphabetical list of keywords for this document, consult the next section (page 93).

Keyword	Description
entire	This entire document.
title	The name of this document.
scope	Topics covered in this document.
availability	Where these programs run.
who	Who to contact for assistance.
introduction	Role and goals of this document.
background	DPCS origins and aims.
dpcs-architecture	Structure, inventory of parts.
resource-allocation	Resource-allocation DPCS parts.
rac	Resource-allocation DPCS parts.
acc	accounts
bac	bank-names
bt	bank-times (defunct)
ra	shift-allocations (defunct)
newacct	current-bank
defacct	default-bank
racmgr	manager-daemon
workload-scheduler	Prod. workload scheduler parts.
pwsd	Prod. workload scheduler parts.
pws-daemons	Compares PWSD, PLSD, and BCD.
pwsd	Compares PWSD, PLSD, and BCD.
plsd	Compares PWSD, PLSD, and BCD.
bcd	Compares PWSD, PLSD, and BCD.
pws-utilities	Compares PWS user tools.
palter	Compares PWS user tools.
pexp	Compares PWS user tools.
phold	Compares PWS user tools.
plim	Compares PWS user tools.
prel	Compares PWS user tools.
prm	Compares PWS user tools.
psub	Compares PWS user tools.
operating-features	How DPCS behaves in practice.
status	Explains allowed job STATUSes.
status-interpretation	Ambiguities, warnings, PSTAT -m.
status-list	Alphabetical status explanations.
class	Explains allowed job CLASSes.
run-properties	Alphabetical PSTAT run-report fields.
job-limits	Bank and user resource partition limits.
environment-variables	Explains DPCS en. var. roles.
comments	Comment removal, shell implications.
shells	Comment removal, shell implications.
job-scheduling	How DPCS schedules jobs.
order	Schedule-precluding conditions, in order.
algorithm	Job scheduling algorithm.
output-truncation	DPCS standard-output limits.
used-resources	Reporting job memory and time used.

log-files	DPCS system logs for debugging.
dfs	How DFS, DCE interact with batch.
nonshareable-resources	Managing nonshareable resources.
expedite-features	Special PALTER features to expedite jobs.
expediting-jobs	How to expedite jobs with PALTER, PEXP.
exempting-jobs	How to exempt jobs with PALTER.
forcing-priorities	How to force job priorities with PALTER.
lrmmgr-permissions	Assigning PALTER permissions with LRMMGR.
fair-share	Fair-share job scheduling explained.
fair-share-definitions	Share, active user terms defined.
shares	Role, consequences, assignment of "shares."
active-users	Role, criteria for "active users."
normalization	Share normalization algorithm.
usage-decay	Usage decay half-life algorithm.
priority	Fair-share priority algorithm, results.
job-scheduling-1	Scheduling and fair-share priority.
priority-service	Graceful priority-service startup.
warnings	Ways to be warned about priority ser.
library-calls	LIBPCS warning-support functions.
pcsgetresource	Reports impending stop time.
lrmgetresource	Reports impending stop time.
pcssig-register	Requests signal if stop impending.
lrmsig-register	Requests signal if stop impending.
pcswarn	Enables a stop-warning variable.
lrmwarn	Enables a stop-warning variable.
pcsstatus	Error conditions in *pcsstatus.
warn-examples	Sample uses of stop-warning tools.
poll-warning	Code examples using PCSWARN.
signal-warning	Code examples using PCSSIG_REGISTER.
admin-examples	Examples using LRMMRG.
checkpointing	Chkpt. instructions and examples.
checkpoint-overview	Chkpt. alternatives compared.
condor-checkpoint	Condor automatic chkpt. on Compaqs.
program-checkpoint	Program-generated chkpt. tips.
checkpoint-script	Script for restart after chkpt.
index	The structural index of keywords.
a	The alphabetical index of keywords.
date	The latest changes to this document.
revisions	The complete revision history.

Alphabetical List of Keywords

Keyword	Description
a	The alphabetical index of keywords.
acc	accounts
active-users	Role, criteria for "active users."
admin-examples	Examples using LRMMRG.
algorithm	Job scheduling algorithm.
availability	Where these programs run.
bac	bank-names
background	DPCS origins and aims.
bcd	Compares PWSD, PLSD, and BCD.
bt	bank-times (defunct)
checkpoint-overview	Chkpt. alternatives compared.
checkpoint-script	Script for restart after chkpt.
checkpointing	Chkpt. instructions and examples.
class	Explains allowed job CLASSes.
comments	Comment removal, shell implications.
condor-checkpoint	Condor automatic chkpt. on Compaqs.
date	The latest changes to this document.
defacct	default-bank
dfs	How DFS, DCE interact with batch.
dpcs-architecture	Structure, inventory of parts.
entire	This entire document.
environment-variables	Explains DPCS en. var. roles.
exempting-jobs	How to exempt jobs with PALTER.
expedite-features	Special PALTER features to expedite jobs.
expediting-jobs	How to expedite jobs with PALTER, PEXP.
fair-share	Fair-share job scheduling explained.
fair-share-definitions	Share, active user terms defined.
forcing-priorities	How to force job priorities with PALTER.
index	The structural index of keywords.
introduction	Role and goals of this document.
job-limits	Bank and user resource partition limits.
job-scheduling	How DPCS schedules jobs.
job-scheduling-1	Scheduling and fair-share priority.
library-calls	LIBPCS warning-support functions.
log-files	DPCS system logs for debugging.
lrmgetresource	Reports impending stop time.
lrmmgr-permissions	Assigning PALTER permissions with LRMMGR.
lrmsig-register	Requests signal if stop impending.
lrmwarn	Enables a stop-warning variable.
newacct	current-bank
nonshareable-resources	Managing nonshareable resources.
normalization	Share normalization algorithm.
operating-features	How DPCS behaves in practice.
order	Schedule-precluding conditions, in order.
output-truncation	DPCS standard-output limits.
palter	Compares PWS user tools.
pcsgetresource	Reports impending stop time.
pcssig-register	Requests signal if stop impending.
pcsstatus	Error conditions in *pcsstatus.
pcswarn	Enables a stop-warning variable.
pexp	Compares PWS user tools.

phold	Compares PWS user tools.
plim	Compares PWS user tools.
plsd	Compares PWSD, PLSD, and BCD.
poll-warning	Code examples using PCSWARN.
prel	Compares PWS user tools.
priority	Fair-share priority algorithm, results.
priority-service	Graceful priority-service startup.
prm	Compares PWS user tools.
program-checkpoint	Program-generated chkpt. tips.
psub	Compares PWS user tools.
pwsd	Prod. workload scheduler parts.
pws-daemons	Compares PWSD, PLSD, and BCD.
pws-utilities	Compares PWS user tools.
pwsd	Compares PWSD, PLSD, and BCD.
ra	shift-allocations (defunct)
rac	Resource-allocation DPCS parts.
racmgr	manager-daemon
resource-allocation	Resource-allocation DPCS parts.
revisions	The complete revision history.
run-properties	Alphabetical PSTAT run-report fields.
scope	Topics covered in this document.
shares	Role, consequences, assignment of "shares."
shells	Comment removal, shell implications.
signal-warning	Code examples using PCSSIG_REGISTER.
status	Explains allowed job STATUSes.
status-interpretation	Ambiguities, warnings, PSTAT -m.
status-list	Alphabetical status explanations.
title	The name of this document.
usage-decay	Usage decay half-life algorithm.
used-resources	Reporting job memory and time used.
warn-examples	Sample uses of stop-warning tools.
warnings	Ways to be warned about priority ser.
who	Who to contact for assistance.
workload-scheduler	Prod. workload scheduler parts.

Date and Revisions

Revision Date	Keyword Affected	Description of Change
05Feb04	warnings	New names added for three functions.
	warn-examples	Two script examples replaced.
	library-calls	Three new keywords added.
	index	Three new keywords added.
10Nov03	index	PCSMGR becomes LRMMGR everywhere, keyword changed.
	status-list	DELAYED, PREEMPTD added.
	class	X (expedited) class clarified.
	run-properties	Six properties added, clarified.
	environment-variables	SLURM env. vars. cross referenced.
	order	Added delay-before-scheduling details.
	used-resources	TIMECHARGED literal added.
	expediting-jobs	Preemption consequences explained.
	title	LCRM added to title.
26Aug03	introduction	Cross ref to SLURM manual added.
20May03	introduction	DPCS officially becomes LCRM.
	background	DPCS officially becomes LCRM.
	algorithm	Four new settable tech-priority attributes.
	expedite-features	New expeditor role formalized.
15Jan03	environment-variables	PSUB_SUBDIR added, PSUB_WORKDIR revised.
13Jan03	expediting-jobs	Now no job limit.
	workload-scheduler	Install mode, gateway node added.
	status-list	NOTIME, RES_WAIT, RUN_SBY, WHOST updated. RM_PEND, WSUBH added.
	class	P obsolete, S clarified.
	algorithm	Short production now obsolete.
	dfs	All DFS/DCE support ended.
	nonshareable-resources	All related DPCS features deactivated.
08Apr02	job-limits	New section on partition limits.
	status-list	Three limit statuses added. Exemptable statuses noted.
	class	New standby (S) class added.
	exempting-jobs	Limit statuses exemptable too.
	index	New keyword for new section.
12Sep01	background	New DPCS function diagram added.
	algorithm	Processor load, historical mem use now part of scheduling.
	expediting-jobs	PSUB now expedites jobs too.

	exempting-jobs	PSUB now exempts jobs too.
	forcing-priorities	PSUB now forces priorities too.
	run-properties	MAXPHYSS, MAXRSS fields added.
	class	Different rates for different classes OK.
	pws-utilities	PSUB, PLIM roles updated.
	dfs	DCE use clarified.
14Mar01	introduction	Cross ref added re managing banks.
	environment-variables	Cross ref added re MPI, Pthreads vars.
	dfs	Cross ref added re new DFS restrictions.
10Jan01	status-list	CPU&TIME status added.
	pws-utilities	job.limits file supplements PLIM.
20Dec00	priority	Fair share formula changed, new terms.
	expedite-features	New sections on expediting, exempting, forcing priorities with PALTER.
	environment-variables	PCS_TMPDIR added, explained.
	pws-utilities	PALTER has new uses.
	class	Fourth (nonstop) class added.
	algorithm	Anticipated cost factor now settable.
	index	New keywords added.
23Oct00	dfs	Need for -noDFS clarified.
19Jun00	status-list	DEFERRED status added.
	class	How class error causes DEFERRED.
	order	DEFERRED status added.
	dfs	-noDFS toggle explained.
10May00	usage-decay	PCSUSAGE replaces older tools.
	rac	PCSUSAGE replaces older tools.
	normalization	Relevant PSHARE line added.
03Mar00	nonshareable-resources	Now works for SCF also.
	status-list	RES_WAIT now for SCF also.
	dfs	Passwordless use clarified.
	entire	All CRAY features deleted.
14Jan00	nonshareable-resources	New section on resource mgmt (OCF).
	status-list	RES_WAIT status added (OCF).
	index	New keyword added.
12Oct99	run-properties	MAXCPUTIME, MAXRUNTIME added. EARLIEST_START, ECOMPTIME added.
	status-list	WCPU redefined, WPRIO added.
	dfs	New URL for DFS info.
09Jun99	priority	Meiko (Tribble) partition deleted.
22Apr99	index	New keyword added.
	used-resources	Updated, cross ref. added.
	run-properties	New PSTAT report section.

05Nov98	index	New keyword added.
	dfs	New section on DFS interactions.
	pcsgetresource	MAXTIME now per task.
01Sep98	scope	Fair-share, checkpoint notes added.
	rac	Fair-share role noted.
	acc	ACC largely disabled now.
	bt	BT defunct now.
	ra	RA defunct now.
	newacct	NEWACCT role limited now.
	defacct	DEFACCT role limited now.
	status	MULTIPLE status on SCF too.
	status-list	Status values now SCF and open.
	log-files	-T now covers 5 days of logs.
	fair-share	Fair-share now on SCF too.
	priority-service	Warnings now on SCF too.
21Apr98	warn-examples	Detailed examples added.
	checkpointing	Checkpointing instructions added.
17Mar98	fair-share	Major new section added.
	used-resources	Highwater PSTAT suboption added.
	bt	Now on SCF only.
	index	Eight new fair-share keywords.
19Feb98	who	POP, DOCGUIDE refs. added.
	introduction	Bank manual cross ref. added.
	rac	Voluntary accounts clarified. Other usage utilities cited too.
	acc	Disabled (open) options noted.
	status	Now compares PSTAT -M and -m.
	status-list	Six new status values added.
	used-resources	New section on memory used.
	log-files	New section on DPCS logs.
	warnings	Warning status updated (open).
	index	Two new keywords added.
04Dec97	priority-service	New section added on priority-service warning calls.
03Nov97	status-list	New subsection for alpha. list.
	status-interpretation	New subsection, PSTAT -m stressed.
17Oct97	environment-variables	Helpful use of PSUB_JOBID noted.
24Sep97	entire	First edition of DPCS Ref. Manual.

TRG (05Feb04)

UCRL-WEB-201535